IMPROVING THE EFFICIENCY AND ROBUSTNESS OF INTRUSION DETECTION SYSTEMS

A Thesis
Presented to
The Academic Faculty

by

Prahlad Fogla

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2007

IMPROVING THE EFFICIENCY AND ROBUSTNESS OF INTRUSION DETECTION SYSTEMS

Approved by:

Wenke Lee, Advisor
School of Computer Science
Georgia Institute of Technology

Hariharan Venkateswaran
School of Computer Science
Georgia Institute of Technology

Mustaque Ahamad
School of Computer Science
Georgia Institute of Technology

Nick Feamster
School of Computer Science
Georgia Institute of Technology

Douglas M. Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved:

ACKNOWLEDGEMENTS

I would like to sincerely thank my research advisor, Dr. Wenke Lee, for his continual support and guidance. He was always eager to discuss new ideas and encouraged me to think independently. He has taught me the art of research for which I will always be thankful to him.

I am grateful to all the committee members for their remarks and comments. Without their insightful suggestions, this thesis would not have been complete. I specially want to thank Dr. Hariharan Venkateswaran for the hours of wonderful discussions on anything theory. I want to thank Prof. Mustaque Ahamad for his guidance through my initial years in the Ph.D program at Georgia Tech.

I want to thank Dr. Emilie Danna for her advice on optimization techniques and various approximation heuristics. I want to thank my colleage, Dr. Roberto Perdisci for his insights on machine learning.

I also want to thank all of my friends for being there in both the good times and the hard times. I am thankful to them for making my stay at Georgia Tech a pleasant experience.

And thanks to my family for instilling the value of education in me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS OR ABBREVIATIONS

DAG	Directed Acyclic Graph.
FSA	Finite State Automaton.
FSM	Finite State Machine.
IDS	Intrusion Detection System.
ILP	Integer Linear Program.
KLT	Karhunen-Loeve Transform.
n-**gram**	Substring of length n. Also called q-gram.
PCA	Principal Component Analysis.
q-**gram**	Substring of length q. Also called n-gram.
SAT	Satisfiability problem.
sFSA	Stochastic Finite State Automaton.

SUMMARY

With the increase in the complexity of computer systems, existing security measures are not enough to prevent attacks. Intrusion detection systems have become an integral part of computer security to detect attempted intrusions. Intrusion detection systems need to be fast in order to detect intrusions in real time. Furthermore, intrusion detection systems need to be robust against the attacks which are disguised to evade them.

We improve the runtime complexity and space requirements of a host-based anomaly detection system that uses q-gram matching. q-gram matching is often used for approximate substring matching problems in a wide range of application areas, including intrusion detection. During the text pre-processing phase, we store all the q-grams present in the text in a tree. We use a tree redundancy pruning algorithm to reduce the size of the tree without losing any information. We also use suffix links for fast linear-time q-gram search during query matching. We compare our work with the Rabin-Karp based hash-table technique, commonly used for multiple q-gram matching.

To analyze the robustness of network anomaly detection systems, we develop a new class of polymorphic attacks called polymorphic blending attacks, that can effectively evade payload-based network anomaly IDSs by carefully matching the statistics of the mutated attack instances to the normal profile. Using PAYL anomaly detection system for our case study, we show that these attacks are practically feasible. We develop a formal framework which is used to analyze polymorphic blending attacks for several network anomaly detection systems. We show that generating an optimal polymorphic blending attack is NP-hard for these anomaly detection systems.

However, we can generate polymorphic blending attacks using the proposed approximation algorithms. The framework can also be used to improve the robustness of an intrusion detector. We suggest some possible countermeasures one can take to improve the robustness of an intrusion detection system against polymorphic blending attacks.

CHAPTER I

INTRODUCTION

Security of computer systems has become a major concern with critical societal infrastructures relying on computers. Considering the complexity of the softwares deployed in network system, intrusion prevention measures like firewalls and cryptographic protocols are deemed insufficient in ensuring the security of the system. Multiple layers of security is required to avoid attacks on the system and to detect possible attacks on the system. Intrusion detection system (IDS) has become an integral part of computer security infrastructure. IDS monitors computers or networks for unauthorized access or activity. IDS detects an attempted, failed or successful intrusion on the system. An IDS can be classified based on the input data or the detection mechanism.

Depending on the data used by an IDS, the IDS can be classified into a network IDS or host-based IDS.

- *Network IDS:* A network IDS analyzes the data transmitted over a network. A network IDS can protect a big network, a LAN, or a single host. Data used by a network IDS includes, packet header data, packet statistics, and application layer payload data. Various network statistics such as rate of incoming packets, rate of failed connections, and average session length can also be used for detection purposes.

- *Host-based IDS:* A host-based IDS is deployed on the host machine to be protected. Various host-related data like commands executed, cpu usage, hard-disk access, memory usage, audit logs and others can be used by a host-based IDS to detect an intrusion. Sequence of system calls executed by a program can also be used for the detection of an intrusion.

Based on the detection mechanism, an IDS can be classified into a misuse IDS or an anomaly IDS.

- *Misuse IDS (or Signature IDS):* A misuse detector uses known patterns of attacks called signatures to catch intrusions. A misuse IDS generates signatures from a given set of attacks. While monitoring, it checks if an attack pattern is present in the monitored data and takes appropriate action when a signature is matched. Hence, misuse IDSs can only detect known attacks.

- *Anomaly IDS:* An anomaly detector records the normal usage patterns of the system. Any system usage which deviates significantly from the normal profile is considered a possible intrusion and an alarm is raised. Unlike a misuse IDS, an anomaly IDS does not require knowledge of attack patterns and thus can possibly detect new attacks.

The two main requirements of an intrusion detector are efficiency and accuracy. In case of an active attacker who is trying to avoid detection, robustness of the IDS is a more practical measure than accuracy. In this dissertation, we analyse and improve on both efficiency and robustness of anomaly detection systems.

1.1 Efficiency

An efficient IDS should be fast and should require less memory. This is essential to detect an attack as early as possible. This is also required so that the IDS can be deployed on a high speed network or a heavily loaded server without significant resource overheads. Misuse IDS and anomaly IDS rely on matching monitored data with attack patterns and normal patterns to detect intrusions. One of the main factors affecting the speed of an intrusion detector is the speed of the matching algorithm it uses. Also, the memory requirement of an IDS is dependent on the space needed to store the patterns used for matching. Thus, it is crucial that the matching algorithm be fast and require minimal memory.

The substring matching problem presented in this dissertation is motivated by a host-based anomaly detection system proposed by Forrest et al. [19]. Normal behavior of a program is observed via its interaction with the underlying operating system, which can be characterized by the sequence of system calls generated by the program. Forrest et al. observed that small sequences of system calls are very consistent throughout different normal executions of a program. We can build a normal profile of a program using the substrings of the system call sequences generated by the sample executions of the program. While monitoring, we can observe the substrings of the system calls generated by the program and check if they match the stored substrings. If a substring does not have a match, an alarm is raised.

The above anomaly detection system requires matching constant-length substrings (called q-grams) present in the observed program and the normal database. The above substring matching problem is called q-gram matching. q-gram matching has been used extensively in many application areas including information retrieval, signal processing, pattern recognition, and computational biology. In computational biology, genomic sequences show high level of matching for small length substrings. In information retrieval, filtration-based approximate string matching algorithms require efficient search of all the q-grams shared by query and text. In some word processors, q-gram matching is performed for approximate matching and finding misspellings. q-gram matching is also used in signal processing for approximate matching.

An anomaly detection system uses a huge set of training normal patterns to avoid false positives. For a huge pattern size, current q-gram matching algorithms become unacceptable. In Section 3, we present a tree-based model for an efficient q-gram matching. The proposed algorithm is extremely memory efficient and is faster than existing algorithms.

1.2 Robustness

The accuracy of an IDS is decided using its *false positive* rate and *true positive* (also called *true detection*) rate. The false positive rate is the fraction of normal data which is incorrectly labeled by an IDS as an attack. True positive rate is the fraction of attack data, successfully detected by an IDS. Ideally, an IDS should detect all the attacks and label all the normal packets as normal. Thus, for an ideal IDS, false positive rate is 0 and true positive rate is 1. Practically, an IDS is said to be accurate if it detects most of the attacks while incorrectly labeling very few normal activities as attacks. That is, an accurate IDS should have a very low false positive rate and a high true positive rate.

Although useful, these two metrics are not sufficient to evaluate an IDS. The false positive rate and the true positive rate are calculated using a test set containing normal data and attack data. In the face of a sophisticated attacker who actively tries to evade the IDS, the IDS may produce much higher false positives or may have much lower true positive rate. For example, in case of an anomaly detection system, an attacker can intelligently craft an attack that is very similar to the normal pattern. The IDS will not be able to detect this attack. These type of attacks are called *mimicry* attacks.

Mimicry attacks have been proposed for host-anomaly detection systems. But no mimicry attacks have been designed for network anomaly IDSs. We present a novel polymorphic attack, called *Polymorphic Blending Attack*, to analyse the robustness of network anomaly IDSs. We show that these attacks are practical for a wide range of network anomaly IDSs. We developed a framework that can be used to analyse the complexity of generating polymorphic blending attacks. We also propose algorithms to automatically generate polymorphic blending attacks. The framework can also be used to improve the robustness of the IDS.

CHAPTER II

RELATED WORK

2.1 String Matching Algorithms

The problem of substring matching has been studied extensively by researchers in several fields. Initial research was mainly focused on complete string matching, required for keyword search in a database. But later the focus shifted to substring matching and approximate string matching. In approximate string matching, a query matches a text with at most k-mismatches.

Suffix trees are used extensively for different string processing problems. A suffix tree can be used for linear time search of a single string in a text. Various algorithms were suggested by Weiner [73], McCreight [37], and Ukkonen [63] for efficient suffix tree generation. Chen and Seiferas [7] proposed a simple technique to efficiently generate a tree to store all the sub-words of a word.

Knuth-Morris-Pratt (KMP) [26] proposed a string searching algorithm. The algorithm preprocesses the query to compute a *shift* function, which is used in the search phase later. The search phase has runtime complexity of $O(m + n)$ where n is the length of text and m is the length of query. Boyer-Moore (BM) [4] suggested a faster algorithm which is now widely implemented for string searching in shell commands (grep) and some editors. The query preprocessing phase of the BM algorithm is of $O(m)$ complexity and the matching phase has the worst case of $O(mn)$ time complexity. But for English text, the average runtime complexity of the BM algorithm is much smaller than the worst case, and is around three times better than the KMP algorithm. Sunday [55] improved on the BM algorithm by improving the BM's shift function and adding a similar shift function used in the KMP algorithm. Sunday's

algorithm has the worst case of $O(m+n)$, and its expected runtime is smaller than the BM algorithm.

String matching algorithm with multiple texts was proposed by Aho-Corasick (AC) [1]. The AC algorithm preprocesses the texts to create a deterministic finite automaton (DFA), which is similar to Trie structure. The algorithm reads the successive characters in the query and makes state transitions based on the next character in the query. The algorithm takes linear time to the length of the query. Coit et al. [10] proposed a fast string matching algorithm, the *AC_BM* algorithm, for intrusion detection. The algorithm stores the texts in a tree similar to Aho-Corasick. Its matching process uses techniques similar to Boyer-Moore.

The above algorithms are based on Boyer-Moore and work well for large queries with a small character size (σ). They do not perform very well for large character sizes or huge texts.

Approximate string matching [38] has also been studied extensively. Landau and Vishkin [33] proposed an $O(kn + km \log m)$ approximate string matching algorithm where k is the number of allowed mismatches. The Lanadu-Vishkin algorithm was improved by the Galil-Ginacarlo algorithm [21], which has $O(kn + m \log m)$ time complexity. Tarhio and Ukkonen [59] proposed a Boyer-Moore algorithm based approach for approximate string matching. Cole and Hariharan [11] presented an $O(n(1 + k^3/m))$ algorithm based on dynamic programming. Wu et al. [75] used DFA for approximate matching in $O(mn/\log s)$ where s is the number of states in DFA. No approximate string matching algorithm is known to have a worst case complexity less than $O(kn)$.

A lot of work has been done on filtration based approximate string matching. Filtration based algorithms first choose a set of positions in the text which are potentially similar to the pattern. Then each pre-selected position is verified for a match using an accurate string matching technique. Karp and Rabin [24] first described a

filtration based approach for exact string matching. Navarro and Baeza-Yates ([39], [40]) proposed an $O(kn \log_\sigma m/\sigma)$ approximate string matching algorithm by improving on the idea developed by Wu and Manber [74]. Later, q-gram based filtration algorithms ([62], [6], [56]) were developed for approximate matching. q-gram filtration approach is based on observation that if a pattern approximately matches a substring of the text, then they share a certain number of q-grams for sufficiently large q. Finding all q-grams shared by the pattern and the text is done by hashing. The algorithm proposed in our paper can be used to find all q-grams efficiently. Chang and Marr [6] proposed filtering based algorithm with an average complexity of $O(\frac{n(k+\log_\sigma m)}{m})$. Furthermore, they proved that this is a lower bound on the average complexity. In practice, filtration based string matching algorithms are the fastest, but their applicability is limited by the error level.

The Rabin-Karp algorithm [24] searches for a substring within a text by hashing. It is suitable for searching multiple patterns of the same length in the text. Suppose we are matching patterns of length q in a text T. First, we compute the hashes of the text patterns and store them in a hash-table. Next we compute and match the hashes of all substrings of the query with the hashes of the text patterns. If two hashes match, then there is a very high probability that the two substrings match. To be completely sure, one can perform a naive matching on the two substrings. In case the hashes do not match, then we can say for sure that the substrings do not match. The main idea of Rabin-Karp is to efficiently compute hashes of successive substrings in constant time by using the hash of the previous substring. Using such a hashing function, matching can be done in expected time of $O(m)$.

QUASAR, proposed by Burkhardt et al. [5], finds all the substrings in a text which has a maximum of k mismatching q-grams as in query. *QUASAR* uses a suffix array to retrieve the positions of any given q-gram in the text. Searching of a given q-gram is done using a hash-table. The preprocessing phase of *QUASAR* takes $O(n \log n)$

time. The matching of a query string takes $O(nm)$ time. The space complexity of the matching algorithm is $O(5n + qS)$, where S is the number of unique q-grams in text. The high complexity of *QUASAR* is because it tries to find all the positions in the text that have a possible matching alignment with the query string.

All of the above string matching algorithms are not suitable for multiple q-gram matching. The best known string matching algorithm takes $O(\frac{n \log_\sigma m}{m})$ expected time. When applied to multiple q-gram matching, the algorithms have an average complexity of $O(\frac{n \log_\sigma q}{q} m)$. This is not acceptable for situations where text size is huge. When applied to q-gram matching problem, Aho-Corasick algorithm takes $O(qm)$ time. The Rabin-Karp algorithm works in linear time of the query size but needs lot of space to store the hash-table. Furthermore, it needs to create separate hash-tables for different values of q.

2.2 Intrusion Detection Systems

Transforming attack packets to avoid detection is a common practice among attackers. Attackers can exploit the ambiguities present in the traffic stream to transform an attack instance to another so that an IDS is not able to recognize the attack pattern. IP and TCP transformations ([23, 44]) techniques are used to evade NIDS that analyzes TCP/IP headers. Vigna et al. [66] discussed multiple network, application and exploit layer (shellcode polymorphism) mutation mechanisms. A formal model to combine multiple transformations was presented by Rubin et al. [48]. Multiple tools such as Fragroute [54], Whisker [45], and AGENT [48] are available that can perform attack mutation.

Code polymorphism has been used extensively by virus writers to write polymorphic viruses. Mistfall, tPE, EXPO, and DINA [57, 76] are some of the polymorphic engines used by virus writers. Worm writers have also started using polymorphic engines. ADMmutate [32], PHATBOT [22], and JempiScodes [50] are some of

the polymorphic shellcode generators commonly used to write polymorphic worms. Garbage and NOP insertions, register shuffling, equivalent code substitution, and encryption/decryption are some of the common techniques used to write polymorphic shellcodes.

Numerous approaches have been proposed to detect polymorphic attacks. In [61], Toth et al. proposed a technique to locate the presence of executable shellcode inside the payload. They used abstract execution of network flows to find the *MEL* (Maximum Executable Length) of the payload. The flow is marked suspicious if its *MEL* is above certain length. Chinchani et al. [8] performed fast static analysis to check if a network flow contains exploit code. STRIDE [2] focuses on detecting polymorphic sleds used by buffer overflow attacks. In [29], Kruegel et al. used structural analysis of binary code to find similarities between different worm instances. Using a graph coloring technique on a worm's control flow graph, this approach is able to accurately model the structure of the worm. Given a set of suspicious flows, Polygraph [41] generates a set of disjoint invariant substrings that are present in multiple suspicious flows. These substrings can then be used as a signature to detect worm instances. In a recent work, Perdisci et al. [42] proposed an attack on Polygraph [41] where noise is injected into the dataset of suspicious flows so that Polygraph is not able to generate a reliable signature for the worm. Shield [69] uses transport layer filters to block the traffic that exploits a known vulnerability. Filters are exploit-independent, and vulnerabilities are described as a partial state machines of the vulnerable application. In [9], Christodorescu et al. proposed an instruction semantics based worm detection technique. The proposed approach can detect code polymorphism that uses instruction reordering, register shuffling, and garbage insertions. It is worth noting that unless the attacker combines the polymorphic blending attack proposed in this paper with other evasion techniques, the approaches cited above [2, 8, 9, 29, 41, 61, 69] may be able to detect the attack.

Several attack specification languages have been developed by researchers to represent the attack signatures efficiently and dis-ambiguously. NETSTAT [15] represents the attack events using a state based machine called STATL. Snort [46] represents a signature using regular expressions consisting network bytes. It also uses other packet attributes. Liang et al. [34] presented extended FSA based attack signatures. GARD [47] is a tool for regular language based generation of attack instances. It is observed that many of the proposed signatures are based on regular grammar or state machine.

A number of attacks aimed at evading Host-based anomaly IDS have been developed. Wagner et al. [68] and Tan et al. [58] presented mimicry attacks against the stide model [20] developed by Forrest et al. The main idea behind these mimicry attacks was to inject dummy system calls into an attack sequence to make the final system call sequence look similar to the normal system call sequence. As a defense against mimicry attacks as well as other *impossible path* attacks [17, 67], more advanced detection approaches (e.g., [16, 17]) were proposed, which use call stack information along with the system call sequences. Recently, a more sophisticated mimicry attack was proposed by Kruegel et al. [28], which can evade most system call based anomaly IDS.

Several application payload-based anomaly IDS [30, 35, 36] have been proposed which monitor the payload of a packet for anomalies. NETAD and LERAD [35, 30] models the first few bytes of the application layer headers of a packet using a set of rules. In [31], Kruegel et al. proposed six different models, namely, length, character distribution, probabilistic regular grammar, token set, attribute presence, and attribute order for detection of `http` attacks. They modeled different `http` attributes like `URL` path, `SQL` query string, etc. Sekar et al. [51] presented an anomaly detection system based on network protocol specifications. Specification is defined using extended finite state automaton. Statistical features based on the state transitions

are monitored to detect anomalies.

PAYL, proposed by Wang and Stolfo [71], records the average frequency of occurrences of each byte in the payload of a normal packet. A separate profile is created for each port and packet length. In their recent work [72], the authors suggested an improved version of PAYL that computes several profiles for each port. At the end of the training, clustering is performed to reduce the number of profiles. They proposed that instead of byte frequency, one can also use an q-gram model in a similar fashion. One main drawback of the system is that they do not consider an advanced attacker, who may know the IDS running at the target and actively try to evade it. In this paper we provide strong evidence that such byte frequency based anomaly IDS are open to attacks and may be easily evaded.

[3] raises the doubts on the security of machine learning based IDS in the face of a determined resourceful attacker. It discussed the possible manipulation of the IDS training process so that the normal space of IDS is gradually moved to include attack packets. Polymorphic blending attack takes a different approach and modifies the attack instance to move it closer to the normal space. CLET [13], an advanced polymorphic engine, comes closest to our polymorphic blending attack. It performs spectrum analysis to evade IDS that use data mining methods for worm detection. Given an attack payload, CLET adds padding bytes in a separate *cramming bytes* zone (of a given length) to try and make the byte frequency distribution of the attack close to the normal traffic. However, the encoded shellcode (using XOR) in CLET may still deviate significantly from the normal distribution and the obtained polymorphic attack may be detected by the IDS. A preliminary work by Kolesnikov et al. [27] introduced and cursorily explored polymorphic blending attacks. In this paper we present a systematic approach for evading byte frequency-based network anomaly IDS, and provide detailed analysis of the design, complexity and possible countermeasures for the polymorphic blending attacks. We also show that our polymorphic

blending technique is much more effective than CLET in evading byte frequency-based anomaly IDS.

CHAPTER III

EFFICIENT APPROXIMATE STRING MATCHING ALGORITHM

3.1 Introduction

Given a text string, T, and a query string, Q, the problem of q-gram matching is to find all the substrings of length q in the query which are also present in the text. This problem can be easily extended to multiple text strings. Since the set of normal or attack patterns needs to be extensive, there is usually a huge number of text patterns in intrusion detection. The length of the query is much smaller than the text.

Existing string or substring matching algorithms are not suitable for the q-gram matching problem and do not have good runtime efficiency. The expected run-time complexity of the best existing string matching algorithm is sub-linear to the length of the text. They do not apply well to multiple q-gram matching with a huge text size. Some string matching algorithms with multiple texts have complexity of the length of the query. When used for multiple q-gram matching, they take $q \times m$ time, where m is the length of query.

A simple solution for the problem is to record all the unique q-grams present in the text and store them in a hash-table. While matching, we can get each of the q-grams in the query and check if it is present in the hash-table. If the calculation of the hash function takes time t, then the total time to match the query will be $O(mt)$ where m is the length of the query. The Rabin-Karp algorithm follows a similar idea and presents an efficient method to calculate hashes. The Rabin-Karp algorithm is the only existing algorithm that we are aware of which works well for the given problem. We present a new q-gram matching algorithm that is more efficient

than the Rabin-Karp algorithm. Although our work was motivated by applications in intrusion detection, the algorithm is general and is applicable to other domains.

The solution we propose makes use of interpretability and efficiency of the tree structure to develop a linear time algorithm for the q-gram matching problem. In the preprocessing step, all the q-grams present in the text are extracted and stored in a tree structure. This tree structure is similar to the Trie structure used for storing dictionaries. We can add suffix links [37] for efficient runtime substring matching. This tree may become very big depending on the character size, the structure of the text, and the value of q. We also propose a tree redundancy pruning algorithm to reduce the size of the tree. The proposed solution has the same linear time complexity as a hash-table but with a smaller constant. It also uses less storage space than a hash-table. In addition, our approach can be used to perform q-gram matching for all the different values of q ($1 \leq q \leq$ Max Tree Depth). A hash-table method, on the other hand, can be used for only one value of q.

3.2 Tree Model

The tree structure provides a general framework for a systematic study of stream data. Trees are computationally efficient for storage, addition, and searching for sets of strings. Tree structures have been used extensively in many applications. For example, decision tree classifiers are successfully used in areas such as character/speech recognition, medical diagnosis, and remote sensing. Prefix trees are used to store and search through dictionaries. Suffix trees are a widely used data structure for text processing.

3.2.1 Notations

In the discussion that follows, we use x, y to represent arbitrary characters and X, Y to represent arbitrary substrings. $X[i, \cdots, j]$ represents a substring of X that starts with the ith character and ends with the jth character. T represents the provided text

and Q stands for the query string. A substring is said to be *accepted* by a substring matching algorithm if it was matched successfully. The substring is *rejected* if no match was found. For any string X, its length is denoted by $|X|$. $X[i]$ denotes the ith character of string X. xX and Xx mean string X is appended with character x at the start and at the end, respectively. $\mathcal{T}$ represents the tree used to store the q-grams of the text T. Figure 1 shows an example tree to store length-4 substrings (or 4-grams) of text 1000011011. $\mathcal{T}_s$ is the tree with suffix link, $\mathcal{T}_p$ is the tree after pruning, and $\mathcal{T}_{sp}$ is the pruned tree with suffix link. The concepts of a pruned tree and a suffix link will be explained in later sections. q denotes the length of the matching substring. $N(l)$ is the number of nodes at depth l of the tree, and N_T is the total number of nodes in the tree. Each node of the tree corresponds to a substring present in the text. For convenience, we use the term *node* $X[i, \cdots, j]$ to denote the tree node corresponding to substring $X[i, \cdots, j]$. For example, node 13 in the Figure 1 is referred as node $T[2, \cdots, 5]$ because it refers to substring $T[2, \cdots, 5]$ (0000). Node $X[i+1, \cdots, j]$ is called *immediate suffix node* of the node $X[i, \cdots, j]$. Node $X[i+l, \cdots, j]$ is called *longest suffix node* of the node $X[i, \cdots, j]$ if none of the nodes $X[i+l', \cdots, j]$, $1 \leq l' < l$, exists in the tree. Two trees $\mathcal{T}_1$ and $\mathcal{T}_2$ are said to be *identical* if, for all nodes in $\mathcal{T}_1$, there is a corresponding node in $\mathcal{T}_2$ with the same path from the root, and vice versa. Tree $\mathcal{T}_1$ is *similar* to tree $\mathcal{T}_2$ if both trees are identical until depth d where $d = min(depth(\mathcal{T}_1),\ depth(\mathcal{T}_2))$.

3.2.2 Tree Structure

A set of strings of length q can be efficiently represented using a depth q tree ($\mathcal{T}$). A node in the tree at depth l is associated with a substring of length l. Also, a link between a node at depth $l-1$ and a node at depth l is associated with the character at position l in the string. Figure 1 shows an example sequence and the corresponding tree for sequence length 4. Constructing such a tree requires $q(|T| - q + 1)$ time.

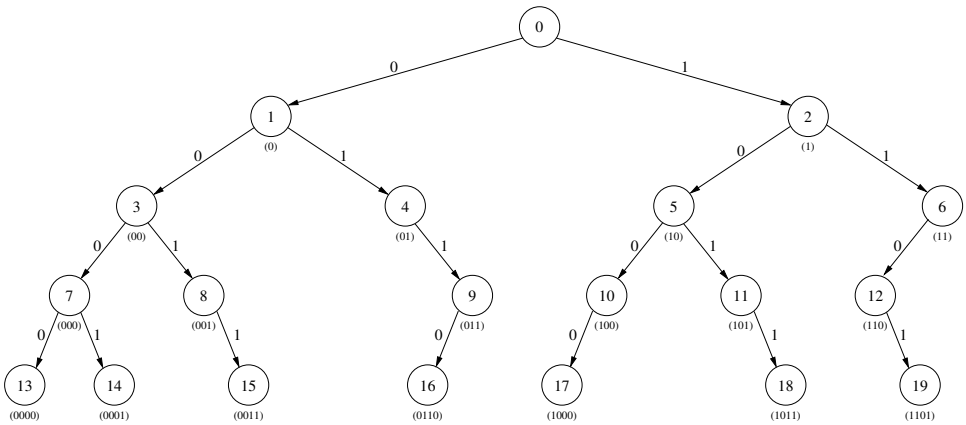


Figure 1: Tree built from text 1000011011 for substrings of length 4. There are 7 length-4 substrings (4-grams) present in the text, namely $\{1000, 0000, 0001, 0011, 0110, 1101, 1011\}$. Each leaf node corresponds to one substring in the set.

Consider the substring matching problem with query $Q[1, \cdots, m]$. For each substring $Q[i, \cdots, i+q-1]$, $1 \leq i \leq |Q|-q+1$ in the query, start from the root node and traverse the tree. At depth l, choose the link corresponding to character $Q[i+l+1]$. If at any depth-l node, there is no link corresponding to character $Q[i+l+1]$, then the substring is not present in the text. If we reach the end of the substring, then we have found a match and the substring is present in the text. A formal description of q-gram matching algorithm using $\mathcal{T}$ is presented in Algorithm 1.

Consider the problem of matching the query $Q = (011000)$ for the tree shown in Figure 1 with $q = 4$. Substrings of length 4 are $(0110, 1100, 1000)$. For substring 0110, we will travel nodes $(0, 1, 4, 9, 16)$, in the given order, and reach the end of the substring. But when we try substring 1100, we will traverse nodes $(0, 2, 6, 12)$ and then stop at node-12 because there is no link for character 0. At this point we can conclude that substring 1100 is not present in the text. Substring 1000 is matched successfully by traversing nodes $(0, 2, 5, 10, 17)$. The above matching process takes at most $O(q)$ steps to match each substring present in the query. Thus, $|Q|$ length query will take $O(q|Q|)$ time.

3.2.3 Tree Redundancy Pruning

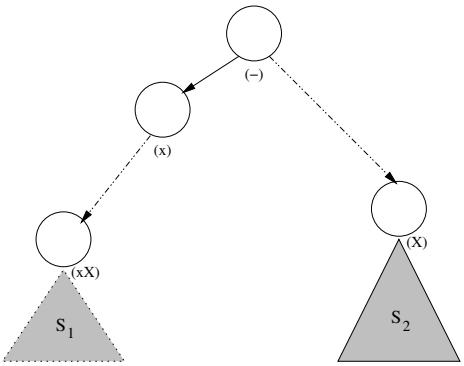
The number of nodes in the tree increases with the number of unique substrings in the text and with the tree depth. The space required to store the tree may become unmanageable for a very large number of unique substrings. We have developed a tree redundancy pruning algorithm which removes redundant nodes present in a tree.

Let us revisit the tree constructed in Figure 1. By looking at node-6, one can infer that substring 11 can be followed by only 01. Now suppose we are matching query ($011xy$). While matching $011x$, we can stop after reaching node-9. Then we can check if x is equal to 0 or not while matching the next substring $11xy$. Thus, we can safely remove the child node (node-16) of node-9. This removal of node-16 should not affect the substring matching capability of the tree. Similarly, the child of node-11 can be removed because the existence of 1 after 101 can be checked while matching the next substring 01?? ('?' matches all characters.) Following the same idea, we can remove all the children of node-3.

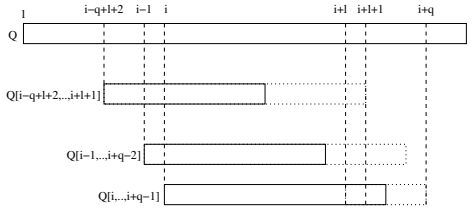
Suppose we construct a tree ($\mathcal{T}$) with depth q as shown in Figure 2(a). If for any node xX at depth l, the $(q-l)$-depth subtree rooted at xX is similar[1] to the subtree rooted at node X, we remove the subtree of node xX and make node xX a leaf node. We repeat this for all the nodes. The final tree ($\mathcal{T}_p$) will be the pruned version of the original tree ($\mathcal{T}$). Pseudo-code for pruning a tree is given in Algorithm A. A pruned version of the tree in Figure 1 is shown in Figure 3.

To prune a tree, for each node, we need to compare the subtree rooted at that node and the subtree rooted at its immediate suffix node. Comparison of two trees takes time linear to the number of nodes in the tree. Thus the total time required to prune the tree is $O(N_T^2)$, where N_T is total number of nodes in the tree.

[1]Recall the definition of tree similarity from section 3.2.1



(a) Pruning Algorithm: Subtrees S_1 and S_2 are similar. Remove S_1.



(b) Marking previous substrings after finding a mismatch.

Figure 2: (a) Tree redundancy pruning algorithm; (b) matching using pruned tree. In both figures, a path up to the solid and dash-dot lines are present in the pruned tree. The dotted part is pruned or absent in the tree.

Consider the substring matching problem with query $Q[1, \cdots, m]$. While matching a substring $Q[i, \cdots, i+q-1]$, if we reach the end of the substring $Q[i, \cdots, i+q-1]$, it is successfully matched and is accepted. If we find a mismatch at character $Q[i+l+1]$, $0 \leq l \leq q-2$, (see Figure 2(b)), the substrings $Q[i-q+l+2, \cdots, i+l+1], Q[i-q+l+3, \cdots, i+l+2], \cdots, Q[i, \cdots, i+q-1]$ are marked *rejected*. If any node in the path of $Q[i, \cdots, i+q-1]$ is pruned, we may reach the leaf node before the end of the substring. In such a case we just mark it as *pending* and continue to match the next substrings. In the end, the substrings which are not *rejected* and marked *pending* are accepted. The detailed description of the matching algorithm using $\mathcal{T}_p$ is given in Algorithm 3.

The complexity of the matching algorithm depends on the effect of pruning on the text and also on the query string. The worst case can be $O(q|Q|)$, which is same as the original tree matching algorithm. We will discuss more results on the effect of pruning in section 3.4.2.

Theorem 3.2.1 *Substring matching using $\mathcal{T}_p$ is equivalent to substring matching using $\mathcal{T}$.*

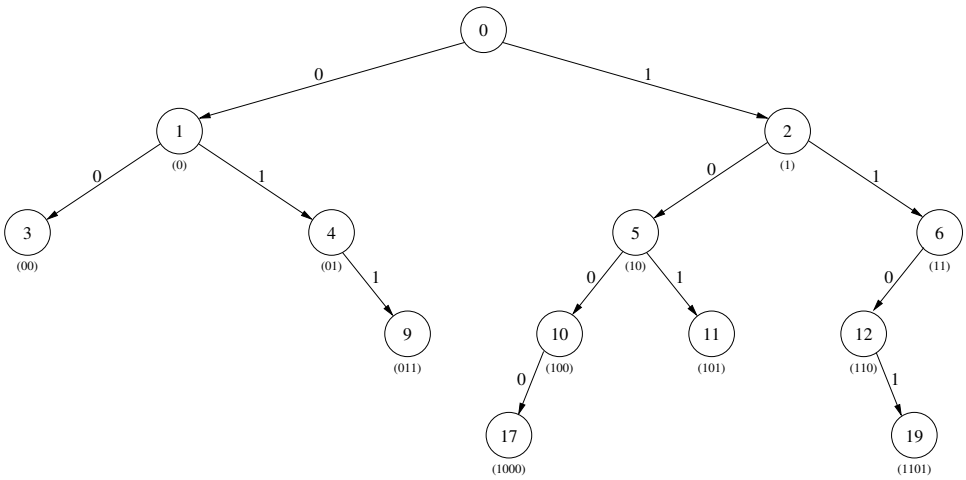
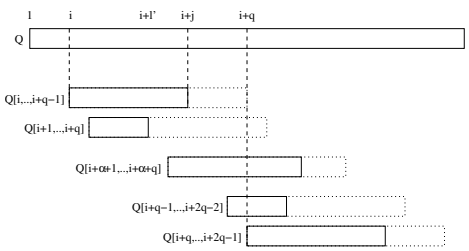


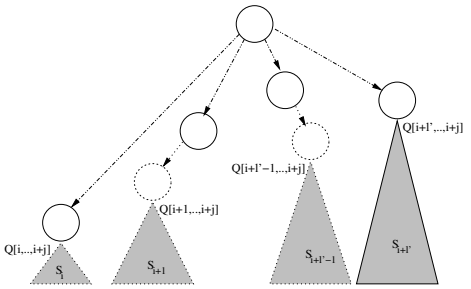
Figure 3: The pruned version of the tree shown in Figure 1. The subtrees of nodes 3, 9, and 11 were removed because they were similar to the subtrees of nodes 1, 6, and 4, respectively.

Proof We shall first prove that if a substring $Q[i, \cdots, i+q-1]$ is accepted by the tree $\mathcal{T}$, then it should also be accepted by the pruned tree $\mathcal{T}_p$. If node $Q[i, \cdots, i+q-1]$ is not pruned, then we will reach the end of the substring while matching it, and it will be accepted by the pruned tree. But if the path to node $Q[i, \cdots, i+q-1]$ is pruned at $Q[i, \cdots, i+j]$ (see Figure 4(a)), then it will be marked *pending*. Consider matching the next $q-1$ substrings ($Q[i+\alpha+1, \cdots, i+\alpha+q]$, $\forall\ 0 \leq \alpha \leq q-2$). We should not find any mismatch before character $Q[i+q]$ because $Q[i+\alpha+1, \cdots, i+q-1]$ are substrings of $Q[i, \cdots, i+q-1]$. Thus $Q[i, \cdots, i+q-1]$ will not be marked as *rejected* because there is no future mismatch in the overlapping substrings, and will be accepted by the pruned tree $\mathcal{T}_p$.

Now we will prove the second part, that if a substring is rejected by tree $\mathcal{T}$, then it will also be rejected by pruned tree $\mathcal{T}_p$. Suppose substring $Q[i, \cdots, i+q-1]$ is not matched successfully in $\mathcal{T}$ at character $Q[i+l+1]$, $1 \leq l \leq q-2$. If the path is not pruned then the substring will also be rejected by the pruned tree. Suppose the path is pruned at $Q[i, \cdots, i+j]$, for some $1 \leq j \leq l$. Suppose character $Q[i+l]$ is not pruned in the subtree of node $Q[i+l', \cdots, i+j]$, $1 \leq l' \leq j-1$ (see Figure 4(b)). All the subtrees



(a) Successful matching of substring $Q[i, \cdots, i+q-1]$. There should be no mismatch before character $Q[i+q]$.



(b) Subtrees $S_i, \cdots, S_{i+l'-1}$ are similar (identical up to length $q-j-1$) to subtree $S_{i+l'}$. All these subtrees except $S_{i+l'}$ are pruned.

Figure 4: Proof of correctness of matching algorithm using pruned tree. The paths up to the solid and dash-dot lines are present in the pruned tree. The dotted part is pruned or absent in the tree.

$S_{i+\alpha}$, $\forall\, 0 \leq \alpha < l'$, are similar to the subtree $S_{i+l'}$. Since path $Q[i+j+1, \cdots, i+l+1]$ is not present in subtree S_i, the path will not be present in subtree $S_{i+l'}$. Thus while checking for substring $Q[i+l', \cdots, i+l'+q-1]$, we will find a mismatch at node $Q[i+l', \cdots, i+l]$. This is because character $Q[i+l+1]$ is not supposed to follow character $Q[i+l]$ in subtree $S_{i+l'}$. At this point substring $Q[i, \cdots, i+q-1]$ will be marked *rejected* along with all the substrings $Q[i+k, \cdots, i+k+q-1]$, $0 \leq k \leq l'$.

3.2.4 Suffix Links

Let us consider the unpruned tree developed in Section 3.2.2. The matching process described for the tree involves duplicated checking. This is due to the overlap of consecutive substrings. We are checking the presence of sub-substring $Q[i+1, \cdots, i+q-1]$ while matching both substrings $Q[i, \cdots, i+q-1]$ and $Q[i+1, \cdots, i+q]$. To remove these redundant checks, we include suffix links in the tree structure. The idea of a suffix link was introduced by McCreight [37] to build a time efficient suffix search tree. Suffix link at each node points to the immediate suffix of the substring corresponding to that node. For example, suffix link at node xX should point to node X. When we match substring xX, we can trace the suffix link and go directly to node X. To check for next substring, we do not need to start again from the root.

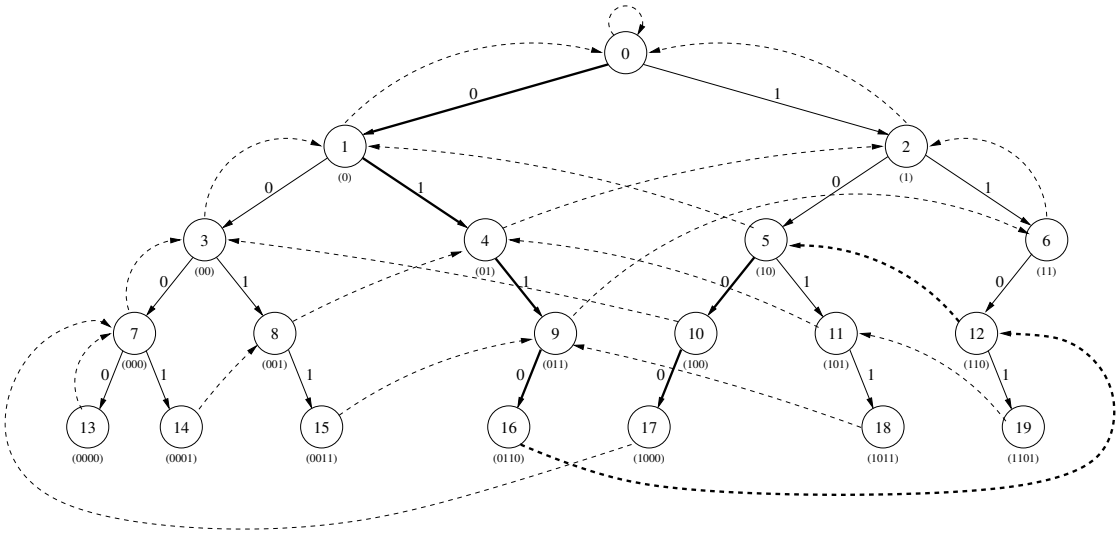


Figure 5: Tree shown in Figure 1 with suffix links. Solid lines are the regular links to child nodes. Dashed lines are suffix links. The path traced by the bold solid and dashed lines, $(0, 1, 4, 9, 16, 12, 5, 10, 17)$, is the path followed while matching query 011000. A mismatch is found at node-12 where there is no child for character 0.

We only need to check the last character of the next substring.

To add suffix links in the unpruned tree, traverse to each node xX in the tree and create a suffix link from node xX to node X. Suffix link at the root node points to itself. Pseudo-code for adding suffix link in an unpruned tree is given in Algorithm 4. Figure 5 shows an example tree with suffix links. Creating a suffix link from a depth-l node requires us to traverse from a root to its suffix. This will take $O(l)$ time. Thus, to construct a suffix link of every node in the tree (with depth q) will take worst-case $O(qN_T)$ time, where N_T is total number of nodes in the tree.

Consider the substring matching problem with query $Q[1, \cdots, m]$. For the first substring, we start from the root node and go down until we find a mismatch or reach a leaf node. If we reach a leaf node $Q[i, \cdots, i+q-1]$ after matching character $Q[i+q-1]$, we accept the substring $Q[i, \cdots, i+q-1]$. To match the next substring $Q[i+1, \cdots, i+q]$, we traverse the suffix link and try to match character $Q[i+q]$.

If we find a mismatch while trying to match character $Q[i+l+1]$ at node $Q[i, \cdots, i+l]$, we reject $Q[i, \cdots, i+q-1]$. To match the next substring $Q[i+$

$1, \cdots, i+q]$, we traverse the suffix link to reach node $Q[i+1, \cdots, i+l]$ and continue the matching process by checking for character $Q[i+l+1]$. We can continue in a similar fashion until we reach the end of the query. A detailed description of substring matching using $\mathcal{T}_s$ is shown in Algorithm 5.

The complete path traced while matching query (011000) is shown in bold in Figure 5. Before proving the equivalence of substring matching with trees $\mathcal{T}$ and $\mathcal{T}_s$, we would like to state some properties that are hold during substring matching using $\mathcal{T}_s$.

Lemma 3.2.1 *If while matching query $Q[1, \cdots, m]$ using $\mathcal{T}_s$, we have processed up to $Q[i]$, then the current node is the longest suffix of $Q[1, \cdots, i]$ present in the tree $\mathcal{T}_s$.*

By Induction For $i = 1$, it is trivial to see that we will be at the longest suffix of string $Q[1]$ after processing $Q[1]$. If character $Q[1]$ is present in the tree, we will progress to node $Q[1]$. If $Q[1]$ is not present in the tree, we will be at the root node. Suppose the lemma holds for some $i = k < m$ and the current node $X = Q[k-l+1, \cdots, k]$ is the longest suffix of $Q[1, \cdots, k]$ present in the tree. Suppose the longest suffix of $Q[1, \cdots, k+1]$ present in the tree is $Q[k-l'+1, \cdots, k, k+1]$, $0 \leq l' \leq l$. In this case, node $Q[k-l'+1, \cdots, k]$ will have child $Q[k+1]$. Also, none of $Q[k-j+1, \cdots, k]$, $l' < j < l$ will have child $Q[k+1]$, otherwise $Q[k-j+1, \cdots, k+1]$ is the longest suffix of $Q[1, \cdots, k+1]$ present in the tree. We want to prove that after processing $Q[k+1]$ we will reach node $Q[k-l'+1, \cdots, k, k+1]$. While trying a match for $Q[k+1]$, starting from node X, we will traverse through $l-l'$ suffix links checking for child $Q[k+1]$. We will find a match only when we reach node $Q[k-l'+1, \cdots, k]$. At this point we will traverse the child node of $Q[k-l'+1, \cdots, k]$ and reach node $Q[k-l'+1, \cdots, k+1]$. Thus, we will be at the longest suffix node of $Q[1, \cdots, k+1]$ after processing $Q[k+1]$. By induction, the lemma should hold for all $i \leq m$. ∎

Theorem 3.2.2 *Substring matching using $\mathcal{T}_s$ is equivalent to substring matching using $\mathcal{T}$.*

Proof To show the equivalence of the two matching algorithms, we will prove that if a substring $Q[i, \cdots, i+q-1]$ in a query is accepted by $\mathcal{T}$, then it will be accepted by $\mathcal{T}_s$, and vice versa. If substring $Q[i, \cdots, i+q-1]$ is accepted by tree $\mathcal{T}$, then there exists a leaf node $Q[i, \cdots, i+q-1]$ in the tree which is also the longest possible suffix of $Q[1, \cdots, i+q-1]$ present in the tree. While matching the query using tree $\mathcal{T}_s$, after we process $Q[i+q-1]$, according to lemma 3.2.1, we will reach the longest suffix node of $Q[1, \cdots, i+q-1]$ present in the tree. From above, this node is the leaf node $Q[i, \cdots, i+q-1]$. Thus, tree $\mathcal{T}_s$ will also accept substring $Q[i, \cdots, i+q-1]$.

Also, if tree $\mathcal{T}_s$ accepts $Q[i, \cdots, i+q-1]$, it means that after processing $Q[i+q-1]$, we will be at depth-q leaf node, say X. According to lemma 3.2.1, X should also be the longest suffix of $Q[1, \cdots, i+q-1]$. But length q suffix of $Q[1, \cdots, i+q-1]$ is $Q[i, \cdots, i+q-1]$. This means that node X corresponds to node $Q[i, \cdots, i+q-1]$. Since node $Q[i, \cdots, i+q-1]$ is present in the tree, substring $Q[i, \cdots, i+q-1]$ will also be accepted by the tree $\mathcal{T}$. ∎

The above matching algorithm is linear to the length of the query. For a traversal of a child link in the tree, we are successfully matching a character and progressing to the next character. For a suffix link traversal from a non-leaf node, we are successfully finding a substring mismatch and we can proceed to the next substring while rejecting the current substring. For a suffix link traversal from a leaf node, we are successfully finding a matching substring and we can proceed to the next substring while accepting the current substring. There are $|Q|$ characters in the query and $|Q|-q+1$ substrings. Thus, the total number of link traversals including child nodes and suffix links is at most $|Q| + (|Q| - q + 1) = O(|Q|)$.

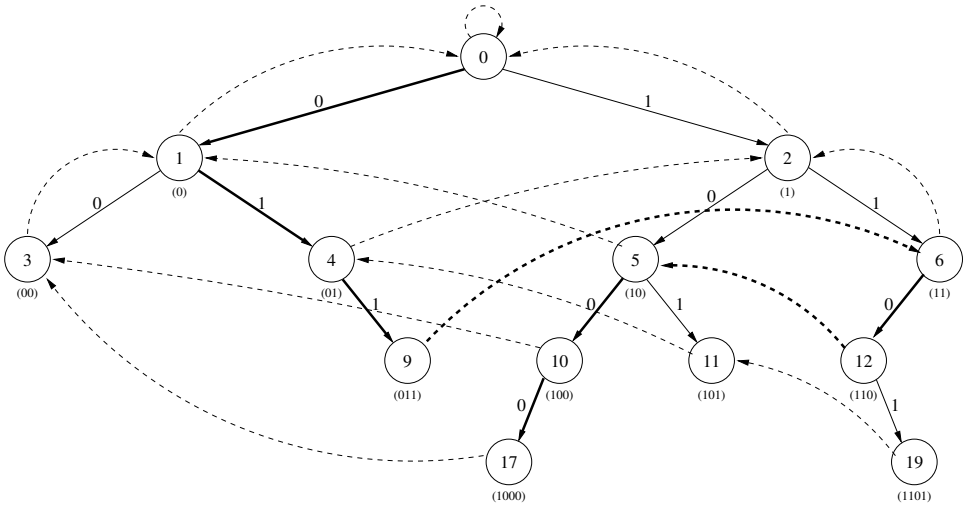


Figure 6: Suffix links added to the pruned tree shown in Figure 3. Solid lines are child links and dashed are suffix links. Node-17's immediate suffix node-7 was pruned from the tree. So it points to immediate suffix node of node-7. Bold lines show the path traced by matching algorithm for query (011000). At node-12 we found a mismatch and substring 1100 was marked *rejected*.

3.2.5 Pruned Trees with Suffix Links

As with the original tree, we would like to add suffix links in the pruned tree to make the matching process faster. But we cannot make a direct link from a node xX to its immediate suffix node X because it is possible that the node X was deleted in the pruning process. Consider a node $Q[i, \cdots, i+j]$ present in the pruned tree. The longest suffix of $Q[i, \cdots, i+j]$ present in the tree is $Q[i+l', \cdots, i+j]$. Suppose we reach node $Q[i, \cdots, i+j]$ while matching a substring. While matching the next $l'-1$ substrings, we will not match beyond $Q[i+j]$ because the path is pruned before $Q[i+j]$ (see Figure 4(b)). Since we have already matched up to $Q[i+j]$ successfully, there is no need to check these $l'-1$ substrings. Thus, when we reach node $Q[i, \cdots, i+j]$, we can directly go to node $Q[i+l', \cdots, i+j]$ and continue checking. Thus, the suffix link at the node $Q[i, \cdots, i+j]$ points to the node $Q[i+l', \cdots, i+j]$.

To add suffix links in a pruned tree, traverse to each node X in the tree and create a suffix link from node X to node Y, where Y is the longest suffix node of X present in the pruned tree. The suffix link at the root node points to itself. Pseudo-code for

adding suffix links in a pruned tree is given in Algorithm 6. Figure 6 shows a pruned tree with suffix links. Substring matching using tree $\mathcal{T}_{sp}$ is very similar to substring matching using tree $\mathcal{T}_s$.

Consider the substring matching problem with query $Q[1, \cdots, m]$. For the first substring, we start from the root node and go down until we find a mismatch or reach a leaf node. If we reach a depth-q node (a leaf node) $Q[i, \cdots, i+q-1]$ after matching character $Q[i+q-1]$, we accept the substring $Q[i, \cdots, i+q-1]$. To match the next substring $Q[i+1, \cdots, i+q]$, we traverse the suffix link and try to match character $Q[i+q]$.

Suppose we reach a leaf node $Q[i, \cdots, i+l]$, $l < q-1$, after matching character $Q[i+l]$. This means that children of this node were pruned during the redundancy pruning. We mark substring $Q[i, \cdots, i+q-1]$ as *pending*. To match next substring, we traverse the suffix link and try to match character $Q[i+l+1]$.

On the other hand if we find a mismatch at node $Q[i, \cdots, i+l]$ for character $Q[i+l+1]$, then the substrings $Q[i-q+l+2, \cdots, i+l+1], Q[i-q+l+3, \cdots, i+l+2], \cdots, Q[i, \cdots, i+q-1]$ are marked *rejected*. After marking the substring, we traverse the suffix link of node $Q[i, \cdots, i+l]$ and continue matching $Q[i+l+1]$ for the next substring $Q[i+1, \cdots, i+q]$. At the end, all the substrings which are not marked *rejected* are *accepted*. The formal description of the matching algorithm is shown in Algorithm 7.

The path traced while matching the query $Q = (011000)$ for the tree in Figure 6 is shown in bold lines. Before proving the equivalence of substring matching with trees $\mathcal{T}_p$ and $\mathcal{T}_{sp}$, we would like to state some properties of substring matching using $\mathcal{T}_{sp}$.

Lemma 3.2.2 *If while matching query $Q[1, \cdots, m]$, we have processed up to $Q[i]$, then the current node is the longest suffix of $Q[1, \cdots, i]$ present in tree $\mathcal{T}_{sp}$.*

by Induction For $i = 1$, it is trivial to see that we will be at the longest suffix of string $Q[1]$ after processing $Q[1]$. If character $Q[1]$ is present in the tree, we

will progress to node $Q[1]$. If $Q[1]$ is not present in the tree, we will be at the root node. Suppose the lemma holds for some $i = k < m$ and the current node $X = Q[k-l+1, \cdots, k]$ is the longest suffix of $Q[1, \cdots, k]$ present in the tree. Suppose the longest suffix of $Q[1, \cdots, k+1]$ present in the tree is $Q[k-l'+1, \cdots, k, k+1]$, $0 \leq l' \leq l$. In this case node $Q[k-l'+1, \cdots, k]$ will have child $Q[k+1]$ and nodes $Q[k-j+1, \cdots, k]$, $l' < j \leq l$ are either pruned or do not have child $Q[k+1]$. While trying a match for $Q[k+1]$, we will travel through suffix links checking for child $Q[k+1]$. We will find a match only when we reach node $Q[k-l'+1, \cdots, k]$. At this point we will traverse child node of $Q[k-l'+1, \cdots, k]$ and reach node $Q[k-l'+1, \cdots, k+1]$. Thus we will be at the longest suffix node of $Q[1, \cdots, k+1]$ after processing $Q[k+1]$. By induction lemma should hold for all $i \leq m$. ∎

Theorem 3.2.3 *Substring matching using $\mathcal{T}_{sp}$ is equivalent to substring matching using $\mathcal{T}_p$.*

Proof To show the equivalence of two matching algorithms, we will prove that if a substring $Q[i, \cdots, i+q-1]$ is marked *accepted* by $\mathcal{T}_p$, then it will be marked *accepted* by $\mathcal{T}_{sp}$, and vice versa. We will also prove that if a mismatch is found while matching substring $Q[i, \cdots, i+q-1]$ by $\mathcal{T}_p$, then a mismatch will be found by $\mathcal{T}_{sp}$, and vice versa. From these two observations, it is implied that if a query is marked first as *pending* in $\mathcal{T}_p$ and accepted (or rejected) later, it will be marked as *pending* in $\mathcal{T}_{sp}$ and accepted (or rejected) later, and vice versa.

If substring $Q[i, \cdots, i+q-1]$ is marked *accepted* by tree $\mathcal{T}_{sp}$, then after processing $Q[i+q-1]$, we will be at depth-q leaf node, say X. According to lemma 3.2.2, X should also be the longest suffix of $Q[1, \cdots, i+q-1]$. But length q suffix of $Q[1, \cdots, i+q-1]$ is $Q[i, \cdots, i+q-1]$. This means that node X corresponds to node $Q[i, \cdots, i+q-1]$. Since node $Q[i, \cdots, i+q-1]$ is present in the tree, substring $Q[i, \cdots, i+q-1]$ will be accepted by tree $\mathcal{T}_p$.

If substring $Q[i,..,i+q-1]$ is marked *accepted* by tree $\mathcal{T}_p$, then there exists a leaf node $Q[i,\cdots,i+q-1]$ in the tree, which is also the longest suffix of $Q[1,\cdots,i+q-1]$ present in the tree. While matching query using tree $\mathcal{T}_{sp}$, after we process $Q[i+q-1]$, according to lemma 3.2.2, we will reach the longest suffix node of $Q[1,\cdots,i+q-1]$ present in the tree. From the above, this node is a leaf node $Q[i,..,i+q-1]$. Thus $\mathcal{T}_{sp}$ will accept substring $Q[i,..,i+q-1]$.

If $\mathcal{T}_{sp}$ finds a mismatch for substring $Q[i,\cdots,i+q-1]$ at node $Q[i,\cdots,i+j]$, then node $Q[i,\cdots,i+j]$ does not have child $Q[i+j+1]$. While matching $Q[i,\cdots,i+q-1]$ using $\mathcal{T}_p$, we will reach node $Q[i,\cdots,i+j]$, and find a mismatch while looking for child $Q[i+j+1]$.

Suppose $\mathcal{T}_p$ finds a mismatch for substring $Q[i,\cdots,i+q-1]$ at node $Q[i,\cdots,i+j]$. While matching using $\mathcal{T}_{sp}$, after we match character $Q[i+j]$, according to lemma 3.2.2, we will be at the longest suffix node of $Q[1,\cdots,i+j]$. Suppose the longest suffix node is $Q[i-l,\cdots,i+j]$, $0 \leq l \leq q-j-1$. Since $Q[i+j+1]$ is not a child of $Q[i,\cdots,i+j]$, $Q[i+j+1]$ can not be a child of $Q[i-l',\cdots,i+j]$, $\forall 0 \leq l' \leq l$. Thus, we will find a mismatch at node $Q[i-l,\cdots,i+j]$ while matching substring $Q[i-l,\cdots,i-l+q-1]$, and traverse the suffix link and again try to match $Q[i+j+1]$. We will continue to find mismatch, traverse suffix links and will eventually reach node $Q[i,\cdots,i+j]$ and still cannot match character $Q[i+j+1]$. Thus, we have found a mismatch for $Q[i,\cdots i+q-1]$. ∎

Corollary 3.2.1 *Substring matching with $\mathcal{T}_{sp}$ is equivalent to substring matching using $\mathcal{T}$.*

Proof This is followed immediately from Theorem 3.2.1 and Theorem 3.2.3. ∎

The above matching algorithm is linear to the length of the query. For every child link traversal, we successfully match a character and progress to the next character. For every suffix link traversal from a leaf node, we mark the current substring as

accepted or *pending* and proceed to the next substring. For every suffix link traversal from a non-leaf node, we find a substring mismatch and proceed to match the next substring after marking the previous overlapping *pending* substrings as *rejected*. Finally, every substring is marked either once, *accepted* or *rejected*, or twice, first *pending* and then *accepted* or *rejected*. There are $|Q|$ characters in the query and $|Q| - q + 1$ substrings. Thus, the total number of link traversals including child and suffix links is at most $|Q| + 2 \times (|Q| - q + 1) = O(|Q|)$.

3.3 String Matching

3.3.1 Exact String Matching

If T is the text and Q is the query, then the exact string matching problem is to find if Q is a substring of T. We will prove that we can use the above q-gram matching for exact string matching. But first, we will prove some results on the number of nodes at a given depth of the tree. We will assume that the text is converted to an infinite length string by adding an infinite number of end symbol (\$) at the end.

Theorem 3.3.1 *For any depth l, $N(l) \leq N(l+1)$, where $N(l)$ is the number of nodes at depth l in tree $\mathcal{T}$.*

Proof Every node X in the tree should have at least one child x. Otherwise, the text string will end after we encounter X in the text. But this is a contradiction because X is assumed to be an infinite length string. Thus, for every node at depth l, we will have at least one node at depth $l+1$. Thus, the number of nodes at depth $l+1$ is at least $N(l)$. ∎

Theorem 3.3.2 $\exists l_0, s.t.\ N(l_0) = N(l_0 + 1)$

Proof Assume $N(l) < N(l+1)$ for all l. For $l = |T| + 2$, where $|T|$ is the length of original text without end symbols added at the end, we will have $N(l) > |T| + 1$.

But the number of unique substrings cannot be more than $|T| + 1$. We have reached a contradiction. Thus, $\exists l_0, s.t. N(l_0) = N(l_0 + 1)$. ∎

Theorem 3.3.3 *If, for some depth* $l_0, N(l_0 + 1) = N(l_0)$*, then*

$$N(l + 1) = N(l) = N(l_0), \ \forall l \geq l_0, \tag{1}$$

Proof If we go down the tree, the number of nodes will increase if and only if there is at least one node with two or more children. Suppose the theorem does not hold for some $l \geq l_0$, that is $N[l + 1] > N[l]$ and node $X[1, \cdots, l]$ at depth l has two children x and x'. Since the substrings $X[1, \cdots, l]x$ and $X[1, \cdots, l]x'$ are present in the tree, their suffix $X[l - l_0, \cdots, l]x$ and $X[l - l_0, \cdots, l]x'$ should also be present in the tree. Then the depth l_0 node $X[l - l_0, \cdots, l]$ should have two children for x and x'. Thus, $N(l_0 + 1) > N(l_0)$. This is a contradiction. Thus we have proved the theorem. ∎

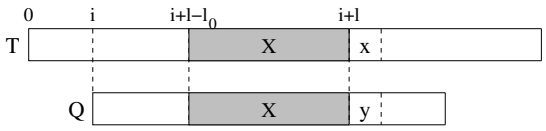


Figure 7: String matching of text T and query Q. X is longest prefix match of Q present in T. Alphabets x and y are different.

Exact String Matching Algorithm Given a text T, make it an infinite length string by adding special end symbols at the end. Build a (l_0+1)-depth tree where the number of nodes stops increasing after depth l_0. Given a query Q, perform q-gram matching with $q = l_0 + 1$. If no mismatch is found then the query is a substring of the text, otherwise it is not.

Theorem 3.3.4 *Query* Q *is a substring of text* T *if and only if no mismatch is found while performing* q*-gram matching with* $q = l_0 + 1$*, where* $N(l_0 + 1) = N(l_0)$*.*

Proof It is easy to see that if the query is a substring of the text, then there will be no mismatch. We need to prove that if Q is not a substring of the text, then we will have at least one mismatching q-gram. Suppose the maximum prefix match of a query in the text starts at i-th position in T and the length of matched prefix is l. If $l \leq l_0$, then substring $Q[i, \cdots, i + l_0]$ is not present in the text and while performing q-gram matching, we will find a mismatch for this substring. The case where $l > l_0$ is shown in Figure 7. Alphabet x and y are different. Since the substring Xx is in the text, it should be present in the tree. Since node X is depth-l_0 node, it cannot have two children, otherwise, $N(l_0) < N(l_0 + 1)$. Thus, substring Xy cannot be present in the tree because otherwise depth l_0 node X will have two children, namely, x and y. So we will find a mismatch at node X in the tree while matching character y from the query. Thus, the exact matching problem is the same as q-gram matching. ∎

3.3.2 Variable Length Matching

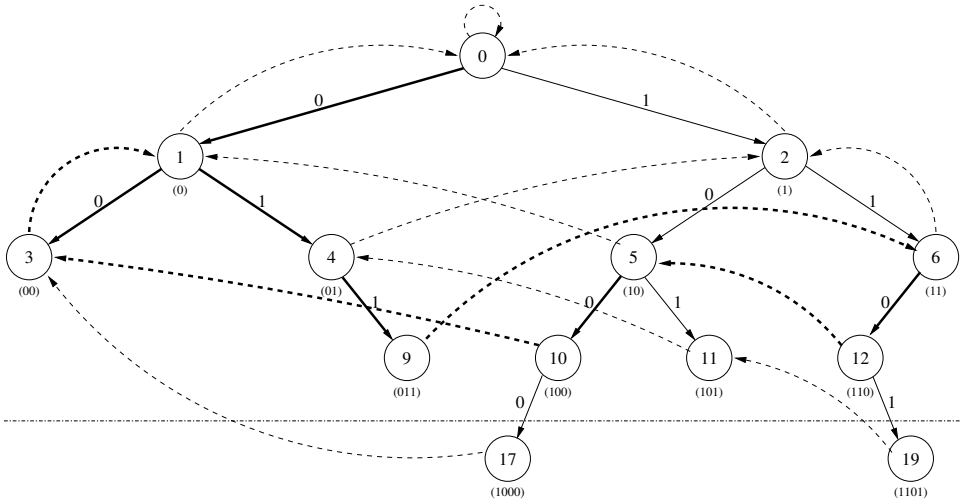


Figure 8: Length-3 substring matching of query 011000 using the tree in constructed in Figure 6. All the nodes at depth 3 are considered leaves and all nodes below them (i.e. node 17 and 19) are considered non-existent. When we reach nodes 12 and 10, we do not try to go any further down to depth-4 node. Instead we traverse the suffix links and continue matching the next character for the next length-3 substring.

The tree model can be used to perform q-gram matching where q is the same as

the maximum depth of the tree. But in many cases, one does not know what sequence length is appropriate. Also, one might need to use different values of q depending on the policy and the situation. For example, in intrusion detection, the value of q is determined by the current required false alarm rate and detection rate. One may want to have larger q for a better detection rate and sometimes a smaller q for a small false alarm rate. We would not want to create a separate tree for each sequence length. This would require more resources and might be cumbersome to maintain.

Given a desired maximum sequence length (say L), we can construct a pruned tree with suffix links as proposed earlier. This tree can be used to perform substring matching of lengths smaller or equal to L. While matching for length $q < L$, we can assume that all the nodes at depth q are leaves and there are no nodes below depth q. Now we can use the same algorithm proposed above for length q substring matching. Consider the problem of matching the text 1000011011 (the original tree is shown in Figure 1) with query 011000 for sequence length 3. All 3-grams in the query are present in the text and we should not find any mismatch. Figure 8 shows the path followed by the algorithm while matching 011000 for sequence length 3.

3.4 Evaluation

We use performance measurements to show the effectiveness of the pruning algorithm. We demonstrate the savings in space because of pruning. We also show the average time complexity of the substring matching using the pruned tree $\mathcal{T}_p$. These two experiments also help us understand the use of tree redundancy pruning to reduce the matching overhead when using pruned tree $\mathcal{T}_p$.

3.4.1 Dataset

For our experimentation purpose, we use the system call sequence datasets made available by University of New Mexico (UNM). These datasets were generated for the stide intrusion detection system developed at UNM [64]. The character size is the

number of different possible system calls, which is equal to 182. In order to collect the data, the given program was started and requests were sent to this program. These requests were either sent by a real user or automatically generated by a program. During the processing of requests, the program makes some calls to the underlying operating system. We can use the *trace* command to record these system calls made by the program in the order they were invoked. The properties of the datasets including their sizes are shown in Table 1. Each dataset consists of multiple sequences generated by multiple runs of the program. As the name suggests, *synthetic sendmail* and *synthetic ftp* datasets were generated at UNM using automatically generated synthetic requests. *Real Sendmail* data was generated by an MIT sendmail server with real user requests. Since the MIT system call trace was very large, only a small fraction (consisting of 5.3 million data points) was used in our experiments. Further information about the dataset may be obtained from [64].

Table 1: System call data

DataSet	Filename	Number of System Calls
Synthetic Sendmail	sendmail.daemon	1.556M
Synthetic FTP	nonself1	180,315
Real Sendmail	MIT	5.23M

To observe the effect of tree redundancy pruning on other datasets, we also generated some random data. In particular, we generated random data with independent Bernoulli distribution and Markovian distribution. Mathematical formulation for Bernoulli distribution is shown in equation 2.

$$s_i = \begin{cases} 1, & \text{with probability } p; \\ 0, & \text{with probability } 1-p \end{cases} \tag{2}$$

where p is the probability of occurrence of 1 in data.

In a markov model, the current output depends on the current state of the model.

For simplicity, we are assuming a simple markov model with binary output. The current output depends on values of the previous two outputs. The sequence satisfies the following stochastic distribution:

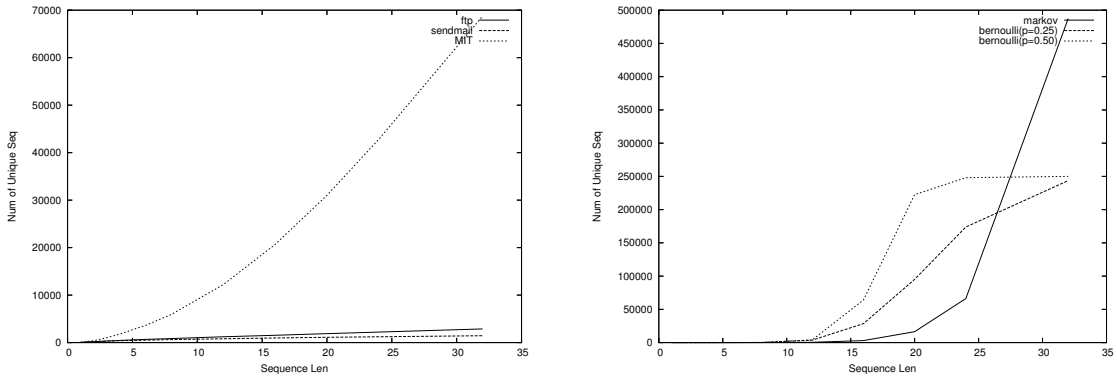
$$s_i = \begin{cases} 1, & \text{if } (s_{i-2}, s_{i-1}) = (0,0); \\ 0, & \text{if } (s_{i-2}, s_{i-1}) = (1,1); \\ 1 \text{ with prob. } p_1, & \text{if } (s_{i-2}, s_{i-1}) = (0,1); \\ 1 \text{ with prob. } p_2, & \text{if } (s_{i-2}, s_{i-1}) = (1,0) \end{cases} \tag{3}$$

where p_1 and p_2 are two prescribed probabilities.

The properties of Bernoulli and Markovian datasets are shown in Table 2. We used two separate Bernoulli datasets with different values of p.

Table 2: Bernoulli and Markovian Dataset

DataSet	Parameters	Number of data points
Bernoulli	p = 0.25	250,000
Bernoulli	p = 0.50	250,000
Markovian	$p_1 = 0.25$, $p_2 = 0.75$	5,000,000



(a) System call Data (b) Bernoulli and Markovian Data

Figure 9: Number of Unique Sequences in Dataset

Figure 9 shows the number of unique sequences in the different datasets. As we expect, the number of unique sequences increases as we increase the sequence length. The MIT dataset shows more diversity and the number of unique sequences increases

rapidly. For the ftp and sendmail dataset, the number of unique sequences remains very low and increases very slowly. For Markovian and Bernoulli dataset, the number of unique sequences increases very fast. This is due to the random nature of these datasets. For Bernoulli dataset, the number of unique sequences stops increasing rapidly after sequence length 24 when it approaches its maximum value of $|T| - q$, where $|T|$ is the number of data points and q is the sequence length.

3.4.2 Experiments

To determine the average time complexity of substring matching using a pruned tree $\mathcal{T}_p$, we calculate the average depth of leaf nodes in the tree. This gives us the expected time taken to match a substring. The average depth of a tree can be computed as

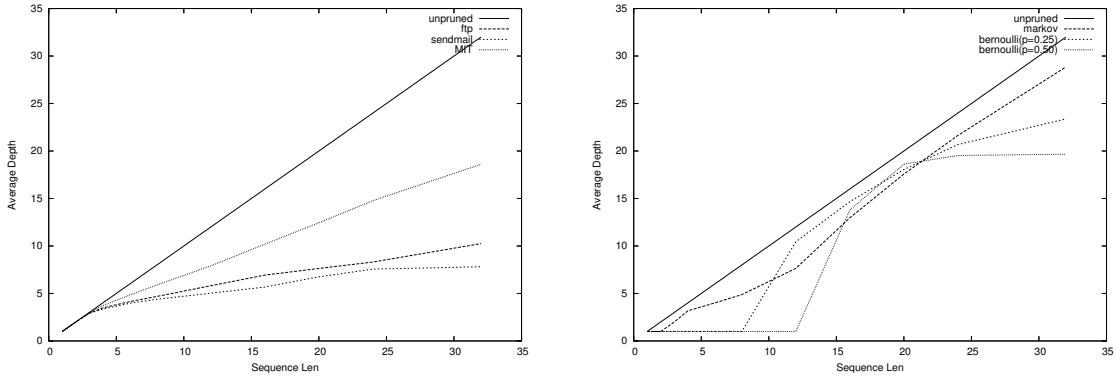
$$Avg.Depth = \frac{\sum_{x \in Leaves} depth(x)}{|Leaves|}, \tag{4}$$

where *Leaves* is the set of all leaf nodes.

The space required to store a tree is directly proportional to the number of nodes and links in the tree. For each node in the tree, there is one incoming link from its parent and one outgoing suffix link. Thus, the number of links is twice the number of nodes. Thus, the space required to store the tree is roughly three times the number of nodes in the tree. In order to compare the space requirements of the unpruned and pruned trees, we calculate the number of nodes for both. We also compared the average space requirements for tree structures with those for hash-tables. Assuming the load factor of hash-table is 1, the minimum space required by the hash-table is $q \times \#(entries\ in\ hash-table)$, where q is the sequence length. The number of entries in the hash-table is equal to the number of unique substrings in the text.

$$Space_{tree} = 3 \times N_T, \tag{5}$$

$$Space_{hash} = q \times \#(\text{unique substrings of length q}) \tag{6}$$



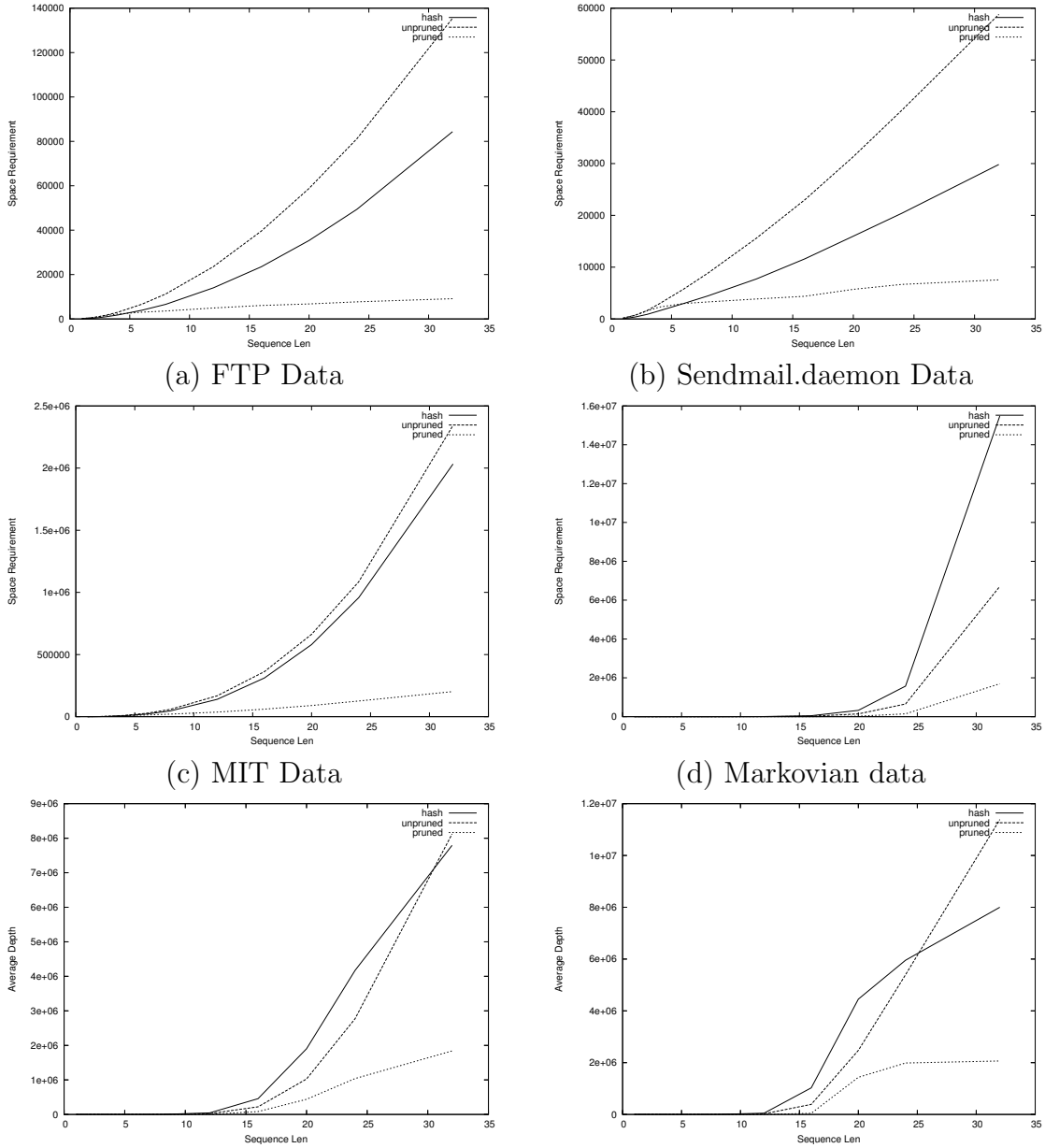
(a) System call Data (b) Bernoulli and Markovian Data

Figure 10: Average depth as a function of sequence length. Solid lines in the figure corresponds to the average depth of the unpruned trees. The average depth of the unpruned trees is equal to the sequence length for all the datasets.

3.4.3 Results

Figure 10 shows the effect of pruning on the average depth of the tree for different sequence lengths. Every leaf node in the unpruned tree is at depth q, where q is the sequence length. Thus, the average depth of the unpruned tree is the same as the sequence length for all datasets. Compared with the unpruned tree, the average depth of the pruned tree increases slowly as we increase the sequence length. Thus, even if we increase the sequence length, there is a small increase in the amount of processing required to match a substring using pruned tree $\mathcal{T}_p$. The effect of pruning is more apparent for ftp and *sendmail.daemon* data as we increase the sequence length. Pruning reduces the average depth by very small value for the Markovian dataset. The subtrees rooted at the bottom of the tree are similar to the subtrees rooted at their suffix nodes. Therefore, they are pruned during the redundancy pruning. Thus, the average depth reduces by a small value. For Bernoulli dataset, pruning works very well for small sequence lengths. This is because for small q, Bernoulli data contains all the possible length-q substrings. Thus, after pruning, the tree reduces to a depth-1 tree with just 3 nodes. For larger sequence lengths, there is not much gain from pruning for Bernoulli data because the data is random in nature. After

sequence length 24 when the number of unique sequences stops increasing rapidly, Bernoulli data again shows significant reduction in average depth.



(a) FTP Data (b) Sendmail.daemon Data

(c) MIT Data (d) Markovian data

(e) Bernoulli data (p=0.25) (f) Bernoulli data (p=0.50)

Figure 11: Space complexity of different datasets.

The space required to store the unpruned and pruned tree is shown in Figure 11. It also shows the minimum space required for a hash-table based matching scheme. For a practical hash-tables with few collisions, we would require more space. The

space requirement of the unpruned tree is slightly more than the hash-table scheme. But for the pruned tree, the space requirement is much less. This means that many of the nodes are being pruned during the pruning process. For larger sequence lengths, pruning reduces the space complexity by an order of 10. Thus, pruning yields a large space saving.

The space requirement for Markovian and Bernoulli datasets is shown in Figure 11 (d), (e), & (f), respectively. For both datasets, space requirement of the unpruned tree is smaller than the hash-table. For Bernoulli dataset, space requirement of hash-table eventually becomes less than the unpruned tree for higher sequence lengths. As for the system call data, pruning shows large savings in space and reduces the space requirement by 4 times. Change in space requirements is more visible for larger sequence lengths.

3.4.4 Comparison with Rabin-Karp

In addition to the above experiments, we also compared our tree model with the Rabin-Karp based hashing technique. For this purpose, we implemented both algorithms in C++. For the given training data, we built a pruned tree with suffix links and performed matching for the testing data. For the Rabin-Karp algorithm, for the given training data, we generated hashes for all the given length substrings present in the data and stored them in the hash-table. While matching the test data, we efficiently generated the successive hashes and checked if that hash value was present in the hash-table. If hash value was found, we considered it as a match. We did not perform further exact matching. If we did not find the value in the table, we got a mismatch. For hashing purposes, we considered a substring as a number in some base. The hash value of the substring was simply the number represented by the substring modulo a big prime. Equation 7 and 8 shows the hashing and rehashing mechanism used in our implementation.

$$hash_{X_i} = (x_i b^{q-1} + \cdots + x_{i+q-1}) \bmod p, \tag{7}$$

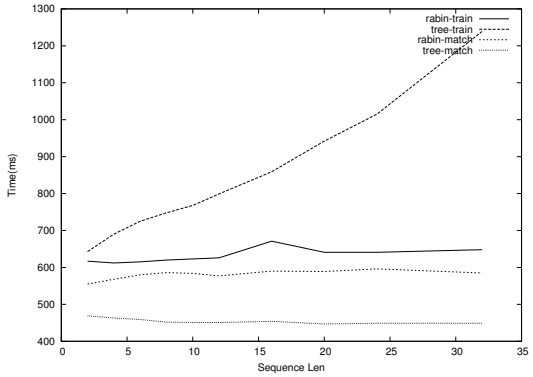
where X is the string, X_i is ith substring, x_i is ith character in the string, p is a random prime, and b is the base.

$$hash_{X_{i+1}} = ((hash_{X_i} - b^{q-1}x_i) \times b + x_{i+q}) \bmod p \tag{8}$$

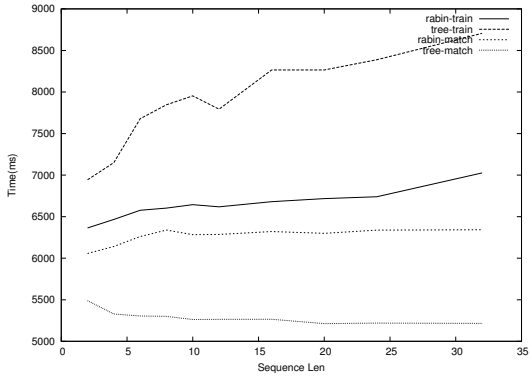
Value of b^{q-1} can be pre-calculated once at the start. The above rehashing technique is efficient and takes constant time to calculate. Thus the first hash can be calculated in time $O(q)$ and further hashes can be calculated in constant time using equation 8. Thus, the calculation of hashes of all the q-grams in the query will take $O(|Q|)$ time, which is linear to the length of the query. We used the same datasets as earlier for comparison. We computed the time taken for preprocessing of the data for both models and compared their runtime overhead for testing the data. For simplicity, we used the same set of data for both training and testing. For efficiency, we set b to 256 and p to a 32-bit random prime.

The time taken to train and perform substring matching is shown in figure 12. For the tree-based algorithm, the training time is calculated as the sum of the time taken to record the unique sequences, build the tree, create suffix links, prune the suffix tree and adjust the suffix links. The training time for tree model is proportional to the square of the size of the tree. The training time of the Rabin-Karp algorithm depends on the time to compute the hash and the time to store the hash in a table. Computing hashes takes constant time. Thus, the training time of Rabin-Karp does not change much as we increase the sequence length. Compared with Rabin-Karp, the tree model takes more time to train, and the time increases with sequence length. This is acceptable because training is an offline process and is performed just once.

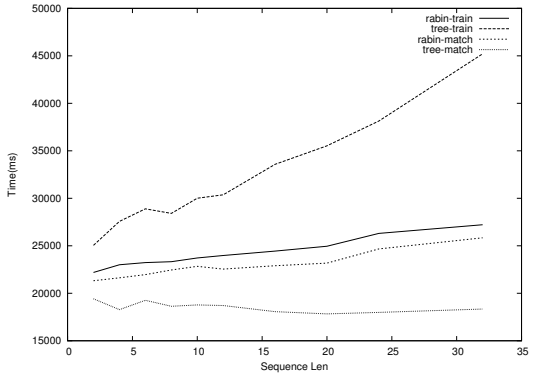
The matching time is almost constant for both algorithms and does not change with sequence length. This is because the matching time of both the Rabin-Karp



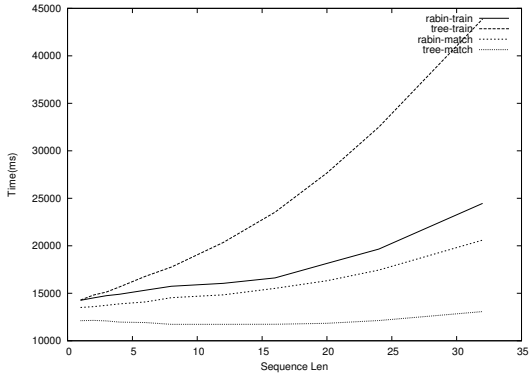
(a) FTP Data



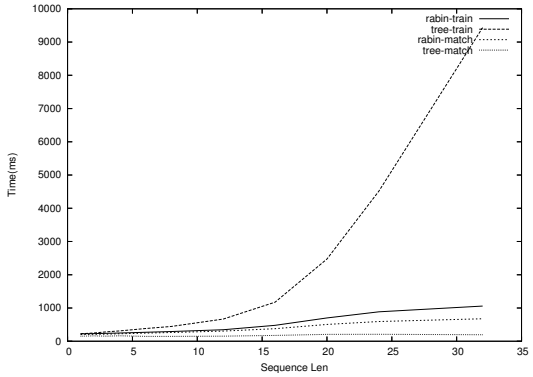
(b) Sendmail.daemon Data



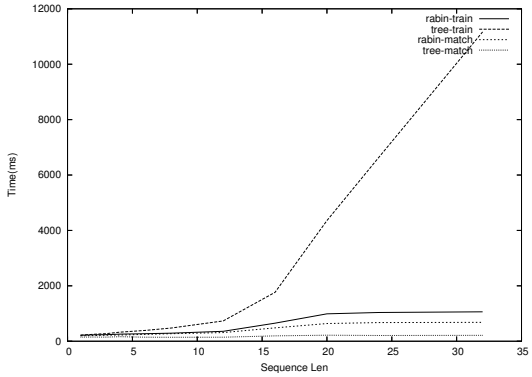
(c) MIT Data



(d) Markovian data



(e) Bernoulli data (p=0.25)



(f) Bernoulli data (p=0.50)

Figure 12: Training and matching time for substring matching.

algorithm and the tree model are independent of the sequence length. The matching time complexity of both algorithms is linear to the length of the query. The tree model takes considerably less time for matching than Rabin-Karp. This is because computation of rehash takes more time than traversing a link in the tree. Since matching is an online process, even a small time saving can be a significant advantage.

3.5 Summary

When applied to q-gram matching problems with huge text size, the computation time required by previous string matching algorithms become unacceptable. The Rabin-Karp algorithm is the only known algorithm which works well for q-gram matching. But the Rabin-Karp algorithm can produce false matches because of the limitations of hashing. Also, for different values of q, the Rabin-Karp algorithm needs to create separate hash-tables. This requires a large amount of space.

We presented a fast q-gram matching algorithm. The algorithm pre-processes the text and stores it in a tree structure that is efficient for storage and addition of new substrings. We also presented a pruning algorithm to reduce the size of the tree. Suffix links were added to the tree structure to facilitate a linear time substring matching algorithm. In addition, we proved that a substring tree of sufficient length can be used to perform exact string matching. Finally, we performed experiments on system call sequence data as well as Bernoulli and Markovian data to show the effect of pruning on runtime and space overheads.

Construction of a tree $\mathcal{T}$ takes $O(q|T|)$ time. In the worst case, redundancy pruning and adding suffix links take $O({N_T}^2)$ and $O(N_T q)$ time, respectively. The matching algorithm using the tree $\mathcal{T}$ takes $O(q|Q|)$ time. q-gram matching with the pruned tree $\mathcal{T}_p$ has the worst case of $O(q|Q|)$ time. But as seen in figure 10, the expected time for matching is very small for system call data. When using trees with suffix links, $\mathcal{T}_s$ or $\mathcal{T}_{sp}$, the matching algorithm is linear to the size of query and does

not depend on the size of the text. For system call data, the space requirement for the unpruned tree with suffix links, $\mathcal{T}_s$, is a little more than the hash-table based q-gram matching system. But for the pruned tree with suffix links, $\mathcal{T}_{sp}$, the space requirement is very modest and is much smaller than that required for the hash-table. Even for large sequence lengths, the space required to store a pruned tree is less than the size of the text. The tree algorithm has a better running time for matching than the Rabin-Karp algorithm. Also, the tree model has an additional advantage in its ability to perform multiple length q-gram matching using the same tree.

CHAPTER IV

ROBUSTNESS OF IDS AGAINST EVASION ATTACKS

In this section we will analyze the robustness of network anomaly detection systems. First we will briefly discuss the attack polymorphism techniques and why anomaly detection system are effective in detection of polymorphic attacks. Then we will look at polymorphic blending attacks. We also present the case study for polymorphic blending attacks using PAYL IDS. Then we present a formal framework for polymorphic blending attacks and analyze it. Then we show experimental results and finally present techniques for improving robustness of IDSs.

4.1 Blending Attacks

4.1.1 Polymorphic Attacks

A *polymorphic attack* is an attack that is able to change its appearance with every instance. Thus, there may be no fixed or predictable signature for the attack. As a result, it may evade detection because most current intrusion detection systems and anti-virus systems are signature-based. Exploit mutation and shellcode polymorphism are two common ways to generate polymorphic attacks. In general, there are five components in a polymorphic attack:

1. Attack Vector: an attack vector is used for exploiting the vulnerability of the target host. Certain parts of the attack vector can be modified to create mutated but still valid exploits. Several attack mutation techniques [23, 44, 66, 48] have been proposed to transform attack vector. Some of the common techniques used for attack mutation are http transformation, multiple requests in same connection, ftp request padding, tcp packet fragmentation, etc. There might

still be certain parts, called the invariant, of the attack vector that have to be present in every mutant for the attack to work. If the attack invariant is very small and exists in the normal traffic, then an IDS may not be able to use it as a signature because it will result in a high number of false positives.

2. Attack Body: the code that performs the intended malicious actions after the vulnerability is exploited. Common techniques to achieve attack body (shellcode) polymorphism include register shuffling, equivalent instruction substitution, instruction reordering, garbage insertions, and encryption. Different keys can be used in encryption for different instances of the attack to ensure that the byte sequence is different every time.

3. Polymorphic Decryptor: this section contains the part of the code that decrypts the shellcode. It decrypts the encrypted attack body and transfers control to it. Polymorphism of the decryptor can be achieved using various code obfuscation techniques such as instruction reordering, instruction substitution, register shuffling, and nop insertions.

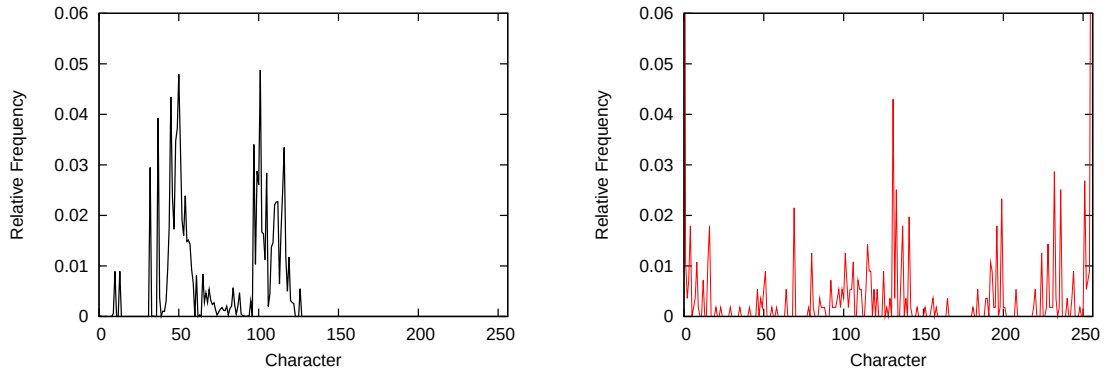
4. Decryption table: used by polymorphic decryptor to decrypt the encrypted attack code.

5. Padding: extra junk data appended in the attack for obfuscation purpose.

4.1.1.1 Anomaly Detection System

The polymorphic attacks generated using different attack mutation and code obfuscation techniques are very effective in evading misuse detection system. Anomaly detection systems [71, 30, 31] that look at the payload of a packet have been proposed to detect the polymorphic attack. A normal `HTTP` request packet predominantly contains printable `ASCII` characters. Figure 13(a) shows the byte frequency distribution of a normal `HTTP` request packet payload. On the other hand, polymorphic attack

instances contain exploit code and input data that are typically not used in normal activities. The exploit code and the data may contain characters that have very low probability of appearing in a normal packet. Figure 13(b) shows the byte frequency distribution of a polymorphic attack packet payload. An anomaly IDS can easily differentiate between polymorphic attack instances and normal packets by looking at the byte frequency distribution of the packet payload. PAYL anomaly IDS [71, 72], which monitors the n-gram (or equivalently q-gram) frequency distribution of packet payload, is shown to be very effective in detection of polymorphic attacks.



(a) Normal HTTP Request Packet (b) Attack Packet

Figure 13: Character distribution of normal and attack packets

4.1.2 Polymorphic Blending Attacks

Clearly, if a polymorphic attack can "blend in" with (or "look" like) normal, it can evade detection by an anomaly-based IDS. Normal traffic contains a lot of syntactic and semantic information, but only a very small amount of such information can be used by a *high speed* network-based anomaly IDS. This is due to fundamental difficulties in modeling complex systems and performance overhead concerns in real-time monitoring. The network traffic profile used by *high speed* anomaly IDS, e.g., `PAYL`, typically includes simple statistics such as maximum or average size and rate of packets, frequency distribution of bytes in packets, and range of tokens at different offsets.

Given the incompleteness and the imprecision of the normal profiles based on simple traffic statistics, it is quite feasible to launch what we call *polymorphic blending attacks*. The main idea is that, when generating a polymorphic attack instance, care can be taken so that its payload characteristics, as measured by the anomaly IDS, will match the normal profile. For example, in order to evade detection by `PAYL` [71, 72], the polymorphic attack instance can carefully choose the characters used in encryption and pad the attack payload with a chosen set of characters, so that the resulting byte frequency of the attack instance closely matches the normal profiles and thus will be considered normal by `PAYL`.

4.1.2.1 A Realistic Attack Scenario

Before presenting the general strategies and techniques used in polymorphic blending attacks, we present an attack scenario and argue that such attacks are realistic. Figure 14 shows the attack scenario that is the basis of our case study. There are a few assumptions behind this scenario:

- The adversary has already compromised a host X inside a network A which communicates with the target host Y inside network B. Network A and host X may lack sufficient security so that the attack can penetrate without getting detected, or the adversary may collude with an insider.

- The adversary has knowledge of the IDS (IDS_B) that monitors the victim host network. This might be possible using a variety of approaches, e.g., social engineering (e.g., company sales or purchase data), fingerprinting, or trial-and-error. We argue that one *cannot* assume that the IDS deployment is a secret, and security by obscurity is a very weak position. We assume IDS_B is a payload *statistics* based system (e.g., `PAYL`). Since the adversary knows the learning algorithm being used by IDS_B, given some packet data from X to Y, the

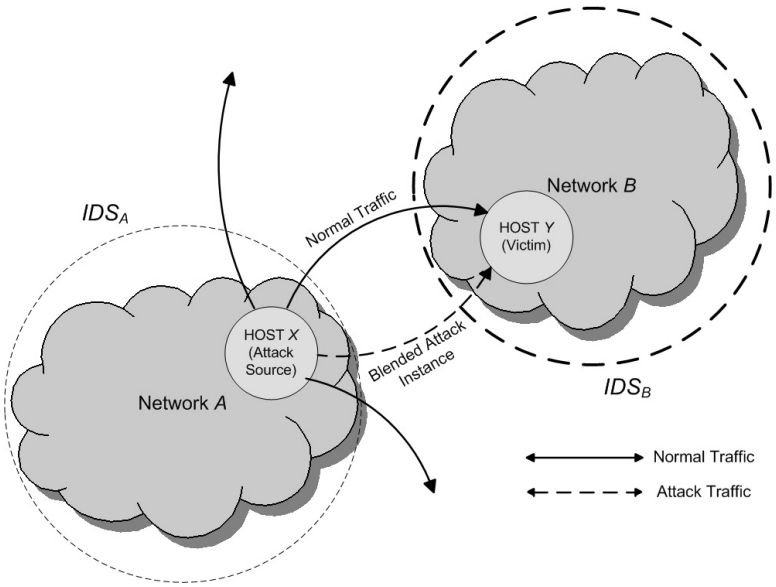


Figure 14: Attack Scenario of Polymorphic Blending Attack

adversary will be able to generate its *own* version of the *statistical* normal profile used by IDS_B.

- A typical anomaly IDS has a threshold setting that can be adjusted to obtain a desired false positive rate. We assume that the adversary does not know the exact value of the threshold used by IDS_B, but has an estimation of the generally acceptable false positive and false negative rates. With this knowledge, the adversary can estimate the error threshold when crafting a new attack instance to match the IDS profile.

We now explain the attack scenario. Once the adversary has control of host X, it observes the normal traffic going from X to Y. The adversary estimates a normal profile for this traffic using the same modeling technique that IDS_B uses. We call this an *artificial profile*. With it, the adversary creates a mutated instance of itself in such a way that the statistics of the mutated instance match the artificial profile. When IDS_B analyzes these mutated attack packets, it is unable to discern them from normal traffic because the artificial profile can be very close to the actual profile in use

by IDS_B. Thus, the attack successfully infiltrates the network B and compromises host Y.

4.1.2.2 Desired Characteristics

Clearly, the key for a polymorphic blending attack to succeed in evading an IDS is to be able to learn an artificial profile that is very close to the actual normal profile used by the IDS, and create polymorphic instances that match the artificial profile. There are other desirable properties. First, the blending process (e.g., with encoding and padding) should not result in an abnormally large attack size. Otherwise, a simple detection heuristic will be to monitor the network flow size. Second, although we do not put any constraint on the resources available to the adversary, the polymorphic blending process should be economical in terms of time and space. Otherwise, it will not only slow down the attack, but also increase the chance of detection by the local IDS (e.g., IDS_A or host-based IDS.) More formally, given a description of the algorithm that the IDS uses to learn and match the normal profile and an attack instance, the time (and space) complexity of the algorithm used to apply polymorphic blending to the attack instance should be a small degree polynomial with respect to the initial attack size. Algorithms that require exponential time and space may not be practical. Since the learning time should be small, the blending algorithm should not require to collect a lot of normal packets to learn the normal statistics.

4.1.3 Steps of Polymorphic Blending Attacks

The polymorphic blending attack has three basic steps: (1) learn the IDS normal profile; (2) encrypt the attack body; (3) and generate a polymorphic decryptor.

4.1.3.1 Learning The IDS Normal Profile

The task at hand for the adversary is to observe the normal traffic going from a host, say X, to another host in the target network, say Y, and generate a normal profile

close to the one used by the IDS at the target network, say IDS_B, using the same algorithm used by the IDS.

A simple method to get the normal data is by sniffing the network traffic going from network A to host Y. This can be easily accomplished in a bus network. In a switched environment, it may be harder to obtain such data. Since the adversary knows the type of service running at the target host, he can simply generate normal request packets and learn the artificial profile using these packets.

In theory, even if the adversary learns a profile from just a single normal packet, and then mutates an attack instance so that it matches the statistics of the normal packet perfectly, the resulting polymorphic blended attack packet should not be flagged as an anomaly by IDS_B, provided the normal packet does not result in a false positive in the first place. On the other hand, it is beneficial to generate an artificial profile that is as close to the normal profile used by IDS_B as possible, so that if a polymorphic blended attack packet matches the artificial profile closely it has a high chance of evading IDS_B. In general, if more normal packets are captured and used by the adversary, she will be able to learn an artificial normal profile that is closer to the normal profile used by IDS_B.

4.1.3.2 Attack Body Encryption

After learning the normal profile, the adversary creates a new attack instance and encrypts (and blends) it to match the normal profile. A straightforward byte substitution scheme followed by padding can be used for encryption. The main idea here is that every character in the attack body can be substituted by a character(s) observed from the normal traffic using a substitution table. The encrypted attack body can then be padded with some more garbage normal data so that the polymorphic blended attack packet can match the normal profile even better. To keep the padding (and hence the packet size) minimal, the substituted attack body should

already match the normal profile closely. We can use this design criterion to produce a suitable substitution table.

To ensure that the substitution algorithm is reversible (for decrypting and running the attack code), a one-to-one or one-to-many mapping can be used. A single-byte substitution is preferred over multi-byte substitution because multi-byte substitution will inflate the size of the attack body after substitution. An obvious requirement of such encryption scheme is that the encrypted attack body should contain characters from only the normal traffic. Although this may be hard for a general encryption technique (because the output typically looks random), it is an easy requirement for a simple byte substitution scheme. However, finding an optimal substitution table that requires minimal padding is a complex problem. We can instead use a greedy method to find an acceptable substitution table. The main idea is to first sort the statistical features in the descending order of the frequency for both the attack body and normal traffic. Then, for each unassigned entry with the highest frequency in the attack body, we simply map it to an available (not yet mapped) normal entry with the highest frequency. This procedure is repeated until all entries in the attack body are mapped. The feature mapping can be translated to a character mapping and a substitution table can be created for encryption and decryption purposes.

4.1.3.3 Polymorphic Decryptor

A decryptor first removes all the extra padding from the encrypted attack body and then uses a reverse substitution table (or decoding table) to decrypt the attack body to produce the original attack code (shellcode).

The decryptor is not encrypted but can be mutated using multiple iterations of shellcode polymorphism processing (e.g., mapping an instruction to an equivalent one randomly chosen from a set of candidates). To reverse the substitution done during blending, the decryptor needs to look up a decoding table that contains the

required reverse mappings. The decoding table for one-to-one mapping can be stored in an array where the i-th entry of the array represents the normal character used to substitute attack character i. Such an decoding table contains only normal characters. Unused entries in the table can be used for padding. On the other hand, storage of decoding tables for one-to-many mapping or variable-length mapping is complicated and typically requires larger space.

4.1.4 Attack Design Issues

4.1.4.1 Incorporating Attack Vector and Decryptor

We discussed in Section 4.1.3.2 that the encryption of the attack body is guided by the need to make the attack packet match the normal statistical profile (or more precisely, the learned artificial profile).

The attack vector, decryptor, and substitution table are not encrypted. Their addition to the attack packet payload alters the packet statistics. The new statistics may deviate significantly from the normal profile. In such a case, we must find a new substitution table in order to match the whole attack packet to the normal profile. First, we take the normal profile and subtract the frequencies of characters in the attack vector, decryptor, and existing substitution table. Next, we find a new substitution table using the adjusted normal profile. If the statistics of the new substitution table is not significantly different from the old substitution table, we use the new substitution table for encryption. Otherwise we repeat the above steps.

4.1.4.2 Packet Length based IDS Profile

If IDS_B has different profiles for packets of different lengths, as in the case of `PAYL`, the substitution phase and padding phase need to use the normal profile corresponding to the final attack packet size. A target length greater than the length of the original attack packet (before polymorphic blending) is chosen at first. The encryption step is then applied and the packet is padded to the target length. If the statistics of the

resulting attack packet is not very close to the normal profile, a different target length is chosen and the above process is repeated. Another strategy is to divide the attack body into multiple small packets and perform the polymorphic blending process for all of them separately.

4.2 Case Study

To demonstrate that polymorphic blending attacks are feasible and practical, we show how an attack can use polymorphic blending to evade the anomaly IDS `PAYL`. `PAYL` has been shown to be effective in detecting polymorphic attacks and worms [71, 72]. For this reason we used `PAYL` in our case study. We used the 2-gram version in addition to the 1-gram version to evaluate how polymorphic blending attack is affected when an IDS uses a more comprehensive model.

4.2.1 Notations

Table 3: Notations

U	Set of all possible distinct alphabets
N	Set of all distinct alphabets in the normal traffic
M	Set of all distinct alphabets in the attack body
u	$\|U\|$
c_n	$\|N\|$
c_m	$\|M\|$
w	Original attack body
$\hat{w}$	Substituted attack body before padding
$\acute{w}$	Substituted attack body after padding
$\|\|s\|\|$	Length of a string s
$f(x)$	Probability of x in normal traffic
$\hat{f}(x)$	Probability of x in $\hat{w}$
$g(y)$	Probability of y in attack body w
T_N	Set of all tuples present in the traffic
T_M	Set of all tuples present in the attack
$S : M \mapsto N$	Mapping from M to N

4.2.2 PAYL

PAYL uses n-gram (or equivalently q-gram) analysis by recording the frequency distribution of n-grams in the payload of a packet. A sliding window of width n is used to record the number of occurrences of all the n-grams present in the payload. A separate model is generated for each packet length. These models are clustered together at the end of the training to reduce the number of models. Furthermore, the length of a packet is also monitored for anomalies. Thus a packet with an unseen or very low frequency length is flagged as an anomaly. $\{f(x_i), \sigma(x_i)\}$ represents the PAYL model of normal traffic, where x_i is the ith gram, which is a character in 1-gram PAYL, and a tuple in 2-gram PAYL. $f(x_i)$ is the average relative frequency of x_i in the normal traffic, and $\sigma(x_i)$ is the standard deviation of x_i in the normal traffic. The anomaly score as calculated by PAYL is shown in Equation 9.

$$score(P) = \sum_i (\mathring{f}(x_i) - f(x_i))/(\sigma(x_i) + \alpha) \tag{9}$$

Here, P is the monitored packet, $\mathring{f}(x_i)$ is the relative frequency of the ith gram x_i in P, and α is a smoothing factor used to prevent division by zero. For convenience we will use the term frequency to denote relative frequency.

We evaluated our polymorphic blending attack with the first version of PAYL as described in [71]. Wang *et al.* [72] proposed some improvements on PAYL in their recent version. We believe that our attack still works for this new version of PAYL. The main improvement of the new version is to use multiple centroids for a given packet length, so that a low false positive rate can be achieved using a relatively low anomaly threshold. In this case, our polymorphic blending attack has to use the same learning algorithm as the new version of PAYL. Furthermore, more normal traffic needs to be used to learn an artificial profile that is close to the actual normal profile. Thus, the effect is that our attack may take a little more time. The new version also matches ingress *suspicious* traffic with egress *suspicious* traffic to find worms. This

feature does not have any effect on our attack because the attack instances blend in with normal.

4.2.3 Evading 1-gram

To evade 1-gram `PAYL`, the frequency of each character in the attack packet should be close to the average frequency recorded during the learning phase. We substitute the characters in the attack packet with the characters seen in the normal traffic, and apply sufficient amount of padding so that the 1-gram frequencies of the resulting packet match the normal profile very closely. We first present analytical results on the amount of padding required to match the substituted attack body with the normal profile perfectly. Then we present a substitution algorithm that uses the padding criteria to minimize the amount of required padding.

In the following sections, we assume that the normal frequency $f(x)$ has already been adjusted for the attack vector, the decryptor, and the decoding table (as discussed in Section 4.1.4.1, these parts need to be accounted for when computing the frequencies of characters to find a suitable substitution).

4.2.3.1 Padding

Let $\hat{w}$ and $\acute{w}$ be the substituted attack body before and after padding, respectively. Let n be the number of distinct characters in the normal traffic. $\|s\|$ denotes the length of a string s, and λ_i denotes the number of occurrences of the normal character x_i in the padding section of the blending packet. Then,

$$\|\acute{w}\| = \|\hat{w}\| + \sum_{i=1}^{c_n} \lambda_i \tag{10}$$

Suppose the relative frequency of character x_i in the normal traffic and the substituted attack body is $f(x_i)$ and $\hat{f}(x_i)$, respectively. Since the final desired frequency of x_i is $f(x_i)$, the number of occurrences of x_i in the blending packet should be $\|\acute{w}\|f(x_i)$.

Thus, λ_i can be defined using the following equation:

$$\lambda_i = \|\acute{w}\| f(x_i) - \|\hat{w}\| \hat{f}(x_i), \quad 1 \leq i \leq c_n \tag{11}$$

Equation 11 can be re-written as,

$$\|\acute{w}\| = \frac{\lambda_i + \|\hat{w}\| \hat{f}(x_i)}{f(x_i)}, \quad 1 \leq i \leq c_n \tag{12}$$

Since $f(x)$ and $\hat{f}(x)$ are relative frequency distributions, $\sum_i f(x_i) = \sum_i \hat{f}(x_i) = 1$. Unless they are identical, there exists some character x_i for which $\hat{f}(x_i) > f(x_i)$. The character x_i is perhaps "overused" in the substituted attack body. It is trivial to see that we need to pad all the characters except the one that is most overused. Let x_k be the character that has highest overuse and δ be the degree of overuse. That is,

$$\delta = \delta_k = max_i\{\delta_i\}, \text{ where } \delta_i = \frac{\hat{f}(x_i)}{f(x_i)}, \quad 1 \leq i \leq c_n \tag{13}$$

Since no padding is required for character x_k, $\lambda_k = 0$. Putting this value in Equation (12) we get:

$$\|\acute{w}\| = \frac{0 + \|\hat{w}\| \hat{f}(x_k)}{f(x_k)} = \delta \|\hat{w}\| \tag{14}$$

The amount of padding required for each character x_i can be calculated by substituting the value of $\|\acute{w}\|$ in Equation (11):

$$\lambda_i = \|\hat{w}\|(\delta f(x_i) - \hat{f}(x_i)) \tag{15}$$

Thus, using the padding defined by the above equation, we can match the final attack packet perfectly to the normal frequency $f(x)$. Furthermore, the amount of padding required by the above equation is the minimum amount that is needed to match the normal profile exactly. Please refer to Appendix B.1 for the proof.

4.2.3.2 Substitution

The analysis in Section 4.2.3.1 shows that the amount of padding can be minimized by minimizing δ, which is $max(\frac{\hat{f}(x_i)}{f(x_i)})$. This in turn means that the objective of the

substitution process is to minimize the resulting δ. There are two possible cases for substitution. The first is when the number of distinct characters present in the attack body (c_m) is less than or equal to the number of distinct characters present in the normal traffic (c_n), i.e. $c_m \leq c_n$. In this case we can perform single-byte encoding, either one-to-one or one-to-many. If $c_m > c_n$, we need to use multi-byte encoding.

4.2.3.2.1 Case: $c_m \leq c_n$ We suggest a greedy algorithm to generate a one-to-many mapping from the attack characters to the normal characters that provides an acceptable solution and is computationally efficient. Our algorithm tries to minimize the ratio δ locally for each substitution assignment.

Let x_i represents a normal character and y_j represent an attack character. Let $f(x_i)$ be the frequency of character x_i in normal traffic and $g(y_j)$ be the frequency of character y_j in the attack body. Let $S(y_j)$ be the set of normal characters to which y_j is mapped. Let $\hat{tf}(y_j) = \Sigma_{x_i \in S(y_j)} f(x_i)$. The probability that y_j is substituted by $x_i, x_i \in S(y_j)$, during substitution is $\frac{f(x_i)}{\hat{tf}(y_j)}$. Thus, the number of occurrences of x_i in the substituted attack body is $\frac{f(x_i) \times g(y_j)}{\hat{tf}(y_j)}$. We then have $\delta_i = \frac{(f(x_i) \times g(y_j)/\hat{tf}(y_j))}{f(x_i)} = \frac{g(y_j)}{\hat{tf}(y_j)}$. Our greedy algorithm tries to minimize this ratio δ_i locally. The substitution algorithm is as follows.

Sort the normal character frequency $f(x)$ and the attack character frequency $g(y)$ in descending order. For the first c_m characters, map y_i to x_i and set $S(y_i) = \{x_i\}$ and $\hat{tf}(y_i) = f(x_i), \forall 1 \leq i \leq c_m$. For the $(c_m + 1)th$ normal character, x_{c_m+1}, find an attack character (y_j) with maximum ratio of $\frac{g(y_j)}{\hat{tf}(y_j)}$. Assign x_{c_m+1} to y_j and set $S(y_j) = \{x_{c_m+1}\} \cup S(y_j)$ and $\hat{tf}(y_j) = \hat{tf}(y_j) + f(x_{c_m+1})$. This is performed for each of the remaining characters until we reach the end of the frequency list $f(x)$. While substituting alphabet y_j in the attack body, we choose a character x_i from the set $S(y_j)$ with probability $\frac{f(x_i)}{\hat{tf}(y_j)}$.

Consider an example where $f(a, b, c) = \{0.3, 0.4, 0.3\}$, attack body $w = qpqppqpq$,

and $g(p,q) = \{0.5, 0.5\}$. According to the above algorithm, initially, b and a are assigned to p and q respectively. At this point, ratio $\frac{g(p)}{\hat{tf}(p)} = 1.25$ and $\frac{g(q)}{\hat{tf}(q)} = 1.66$. So we assign c to q. Thus, p will be substituted by b and q will be substituted by a with probability 0.5 and by c with probability 0.5. Thus, the attack after substitution can be $\hat{w} = cbabbcba$.

In our experiments, we used a simple one-to-one mapping where characters with the highest frequencies in the attack packet are mapped to characters with the highest frequencies in normal traffic. This simple mapping is shown to be sufficient for the blending purpose.

4.2.3.2.2 Case: $c_m > c_n$ We suggest a heuristic based on Huffman encoding scheme to obtain a small attack size after encoding. Given the frequency distribution of the characters in the attack body being encoded, Huffman encoding provides a minimum length packet after encoding. The weights of the nodes in Huffman tree is the sum of the relative frequencies of all its descendant leaf nodes. The weight of a leaf node is the frequency of a given character in the attack body. Every edge in the tree is assigned to a character from the normal profile. In the original Huffman coding the edges of the Huffman tree are labeled randomly. Random labeling of the edges may give us a very large value of δ. We developed a heuristic to assign labels to edges of Huffman tree to find a mapping that gives us a very small δ. Before stating the heuristic, we present the problem of optimally assigning the labels to the edges in Huffman tree:

Given a Huffman tree, assign labels $l(v) \in N$ to the vertices v in the tree, such that after substitution, $\delta = max(\frac{\hat{f}(x)}{f(x)}), \forall x \in N$, is minimum. The constraint on the label $l(v)$ is that if $parent(v_1) = parent(v_2)$, then $l(v_1) \neq l(v_2)$.

We propose a greedy algorithm to find an approximate solution for the above problem. First sort the vertices in descending order of their weight and initialize the

capacity of each character $cap(x_i) = f(x_i), \forall x_i \in N$. Then starting from the leftmost unlabeled vertex v_j, find a character x_i with the maximum $cap(x_i)$ and that is not assigned to any of the direct siblings of v_j. Assign x_i to v_j and reduce the capacity of x_i by the weight of the vertex. Repeat until all the vertices are assigned. The labels generated by the above algorithm are used for the substitution process. An example is explained in Figure 15.

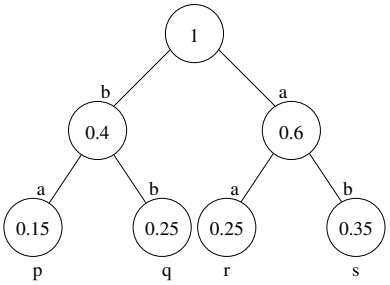


Figure 15: 1-gram multibyte encoding. The frequency of the normal character is $f(a, b) = \{0.5, 0.5\}$. Sorted weights of the nodes are $\{0.6, 0.4, 0.35, 0.25,\ 0.25, 0.15\}$. Using the proposed algorithm we get $S : \{p, q, r, s\} \mapsto \{ba, bb, aa, ab\}$

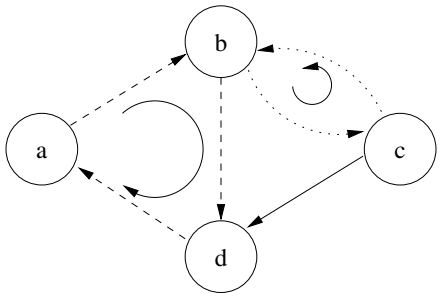


Figure 16: 2-gram multibyte encoding. $e_0 = da$, $e_1 = bc$. $w = 01101010$. $\hat{w} =$ *bdabcbcbdabcbdabcbda*

4.2.4 Evading 2-gram

The 1-gram `PAYL` model assumes that the bytes occurring in the stream are independent. It does not try to capture any information of byte sequencing of the normal traffic. The 2-gram model on the other hand can capture some byte sequencing information. It records the frequencies of all the 2-grams present in the normal traffic. It is

easy to see that by matching 2-grams we are inherently performing 1-gram matching as well.

For 2-gram, the polymorphic blending process needs to match the frequencies of not only all the characters but also all the tuples. Similar to 1-gram substitution, one can either use single-byte encoding or multi-byte encoding for substitution. For single-byte encoding, the goal is to find a one-to-one or one-to-many mapping that ensures that all the tuples in the substituted attack body are also present in normal profile. In Appendix B.2, we show that this is NP-complete for the general case by reducing the well known sub-graph isomorphism problem [12] to the mapping problem. Unlike single-byte encoding, it is possible for an attacker to find a multi-byte encoding scheme that produces only valid 2-grams. Here, we present a viable multi-byte encoding scheme.

4.2.4.1 Multi-byte Encoding

A 2-gram normal profile can be viewed as a Moore machine (FSM) which has a state for each character in N. Every state is a start state and end state. A transition from state v_1 to state v_2 exists if and only if 2-gram v_1v_2 exists in normal profile. This FSM represents the language accepted by the IDS with given 2-gram profile. Strings generated by the FSM contain only normal 2-grams. Characters in an attack body can be mapped to paths in this FSM. For example, suppose the state machine has two cycles reachable from each other. e_1 and e_2 be two edges such that e_1 is present only in the first cycle and e_2 is present only in the second cycle. Given a bit representation of the attack body, we can encode 0 using e_0 and 1 using e_1. We can generate any bit string represented using these two tuples interleaved by other non-informative characters present in the cycles and in the paths between two cycles. Figure 16 shows an example of such an encoding scheme. Such an encoded attack string will have a very large size. We use it to show the existence of an encoding scheme that is able

to match the normal 2-grams. We can generate a more efficient encoding scheme by using the entropy measure of transitions at each state. The complete details of such an encoding scheme are not addressed in this paper. The authors suggest readers to refer to coding theory for more on entropy based encoding.

4.2.4.2 Approximate Single-Byte Encoding

As discussed above, the problem of finding a single-byte substitution is hard for 2-gram. On the other hand, multi-byte encoding may increase the size of the attack packets considerably. We can use a simple approximation algorithm to find a good one-to-one substitution. The algorithm performs single byte substitution in such a way that tuples with high frequencies in the attack packet are greedily matched with tuples with high frequencies in normal traffic.

The details of the algorithm are as follows. First, sort the normal tuple frequencies $f(x_{i,j})$ and the attack tuple frequencies $g(y_{i,j})$ in descending order. Initially, all tuples in the list $f(x_{i,j})$ are marked *unused* and the substitution table is cleared. The frequency list $g(y)$ is traversed from the top. For every tuple $y_{i,j}$ in the sorted attack tuple list, the list $f(x)$ is traversed from the beginning to find an *unmarked* tuple $x_{i',j'}$ so that substituting y_i with $x_{i'}$ and y_j with $x_{j'}$ does not violate any mappings that were already made. The tuple $x_{i',j'}$ is *marked* and the substitution table is updated. The above algorithm is fast and provides consistent reversible matching. The algorithm does not guarantee to provide the best substitution, i.e., the closest distance to the target frequency distribution.

4.2.4.3 Padding

We introduce an efficient padding algorithm that does not provide minimal padding but tries to match the target distribution in a greedy manner. Let $d_f(x_{i,j})$ be the difference between the frequency of tuple $x_{i,j}$ in the normal profile and the substituted attack body. Find a tuple $x_{k,l}$ from the list of normal tuples that starts with the last

padded character (x_k) and that has the highest $d_f(x_{k,m}), \forall 1 \leq m \leq 256$. The second character of the tuple, x_l, is padded to the end of the packet and $d_f(x_{k,l})$ is reduced. This step is repeated until the blending attack size reaches a desired length.

4.2.5 Complexity of Blending Attacks

We now summarize the methods provided above and analyze the hardness of a polymorphic blending attack while keeping the design goals (Section 4.1.2.2) in mind. For 1-gram blending, although finding a substitution that minimizes the padding seems to be a hard-problem and may take exponential time, we have proposed greedy algorithms that find a good substitution that require small amount of padding to perfectly match the normal byte frequency. For 2-gram blending, finding a single-byte substitution that ensures only normal tuples after substitution is shown to be NP-hard (see the proof in Appendix B.2). An approximation algorithm can be used to efficiently compute a substitution that may introduce a few invalid tuples. A multibyte encoding scheme can achieve a very good match with no invalid tuples at the expense of very large attack sizes. An attacker has to therefore consider several trade-offs between the degree of matching, attack size, and time complexity to mount successful blending attacks.

4.2.6 Experiments

In our evaluation, we first established a baseline performance by sending polymorphic instances (generated using the CLET polymorphic engine) of the attack to `PAYL` and verified that all of the instances were detected by the IDS as anomalies. Then, without changing the configuration of `PAYL`, we used our polymorphic blending techniques to generate attack instances to see how well they can evade the IDS.

4.2.6.1 Experiment Setup

4.2.6.1.1 Attack Vector We chose an attack that targets a vulnerability in Windows Media Services (MS03-022). The attack vector we selected exploits a problem with the logging `ISAPI` extension that handles incoming client requests. It is based on the implementation by `firew0rker` [18]. The size of the attack vector is 99 bytes and is required to be present at the start of the `HTTP` request. The attack needs to send approximately 10KB of data to cause the buffer overflow and compromise the system. Our attack body opens a `TCP` client socket to an IP address and sends system registry files. The size of the unencrypted attack body is 558 bytes and contains 109 different characters. During the blending process, we divided our attack into several packets. If our final blending attack after padding does not add up to 10KB, we just send some normal packets as a part of the attack to cause the buffer overflow. The decryptor was divided into multiple sections and distributed among different packets. The attack body was divided among all the attack packets.

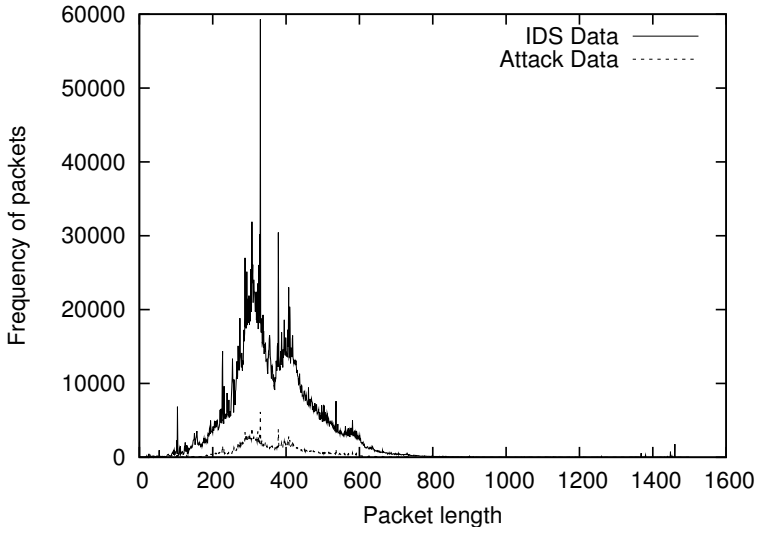


Figure 17: Packet length distribution

4.2.6.1.2 Dataset We collected around 7 days of `HTTP` traffic (4.7M packets) coming to our department's network in November 2004. We used several IDSs, including

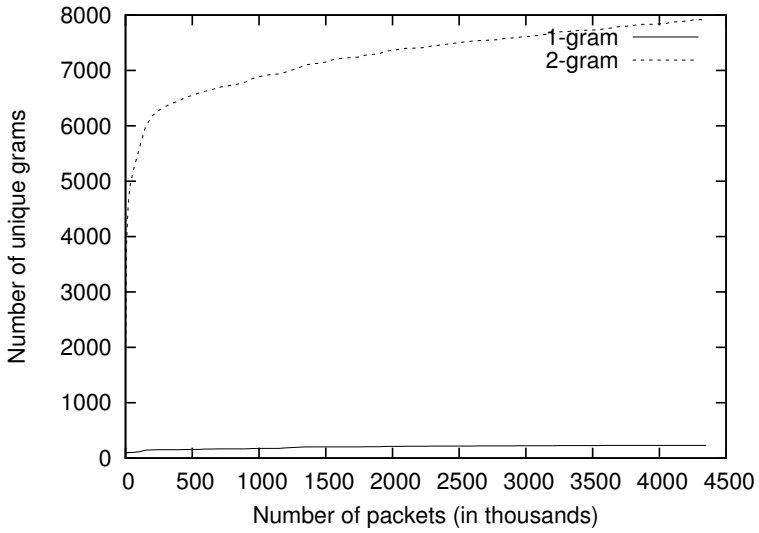


Figure 18: Observed unique 1-grams and 2-grams

Table 4: `HTTP` Traffic dataset

Data Type	Feature	Packet length		
		418	730	1460
IDS Training	Num. of Pkts	16,490	540	1,781
	One Grams	106	90	128
	Two Grams	4,325	3,791	3,903
Attack Training	Num. of Pkts	2,168	82	249
	One Grams	89	86	86
	Two Grams	2,847	2,012	2,196

Snort, to verify that this data contains no known attack. We removed all the packets with no `TCP` payload. We used around 4.3M packets (1.9GB) for IDS training to obtain the IDS normal profiles. A separate profile was created for each `TCP` payload length (or simply packet length). The full payload section of each packet was used to compute the profiles. The last day of the HTTP traffic was made available to the attacker to learn the artificial profile. We also used cross-validation, i.e., randomly picking one of the 7 days for attack training and the rest for IDS training, to verify the results of our experiments.

The packet length distributions in the IDS training dataset and the attack training dataset are shown in Figure 17. Among this packet lengths, we chose three different

lengths to implement the blending attack, namely 418, 730 and 1460. These packets lengths are large enough to accommodate the attack data into a small number of packets. These lengths also occurred frequently in the training dataset. A separate artificial profile was created for each packet length using the attack training data of the same packet length. Thus, we generated three 1-gram models and three 2-gram models for different packet lengths. Table 4 shows the details of the datasets used for the evaluation. The numbers of unique 1-grams and 2-grams in the data are also shown in the table.

4.2.7 Results

Training time of 1-gram and 2-gram `PAYL`: We performed experiments on the training time required to learn the profiles used by `PAYL`. Figure 18 shows the numbers of unique 1-grams and 2-grams observed in `HTTP` traffic stream. Since the numbers of observed 1-gram and 2-gram continue to increase as new packets arrive in the stream, the training of profiles for 1-gram and 2-gram takes a long time to converge. We trained our IDS model using all of the available IDS training data.

Traditional polymorphic attacks: To the best of our knowledge, CLET [13] is the only publicly available tool that implements evasion techniques against byte frequency-based anomaly IDS. For this reason we used CLET as our baseline. As mentioned in Section 2.2, given an attack CLET adds padding bytes in the payload to make the byte frequency distribution of the attack close to the normal traffic. However, CLET does not apply any byte substitution technique (see Section 4.2.3.2). Further, CLET does not address the evasion of 2-gram `PAYL` explicitly. We also generated polymorphic attacks using other well known tools (e.g., ADMutate [32]), and verified that they are less effective than CLET in evading `PAYL`.

We generated multiple polymorphic instances of our attack body using CLET and tested them against `PAYL`. Each attack instance contained one or more attack packets

of given length. Different amount of bytes were crammed (padded) to obtain the desired attack size. Attack training data was used to generate spectral files used for cramming by the CLET engine. A polymorphic attack instance will evade an IDS model if and only if all the attack packets corresponding to the attack instance are able to evade the IDS. Thus, the anomaly score of an attack instance was calculated as the highest of all the anomaly scores (Equation 9) obtained by the attack packets corresponding to the attack instance. Table 5 shows the anomaly threshold setting of different `PAYL` models that result in the detection of all the attack instances. The anomaly thresholds were calculated as the minimum anomaly score over all the attack instances. Using the given thresholds, both 1-gram and 2-gram `PAYL` were successful in detecting all the instances of the attack. Having established this "baseline" performance, we would like to show that our blending attacks can evade `PAYL` even if a lower threshold is used.

Table 5: IDS anomaly threshold setting that detects all the polymorphic attacks sent by the CLET engine

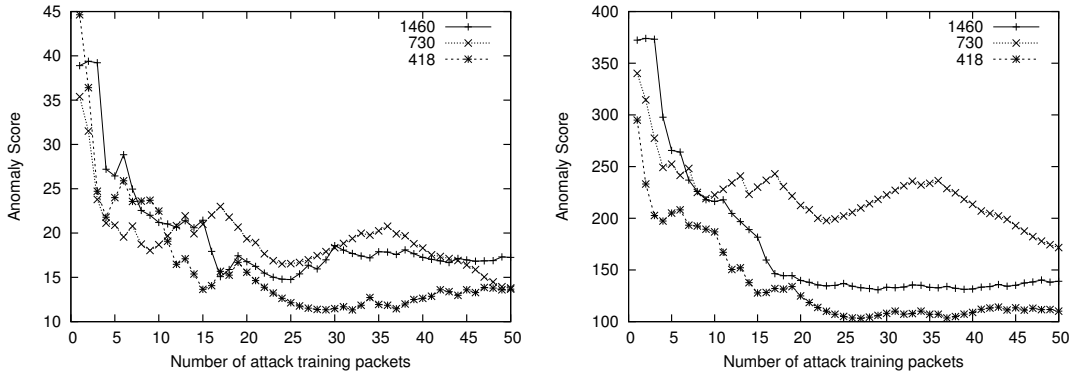
Packet Length	1-gram	2-gram
418	872	1,399
730	652	1,313
1460	355	977

Table 6: Number of packets required for the convergence of attacker's training

Packet Length	1-gram	2-gram
418	8	20
730	8	18
1460	14	40

4.2.7.1 Artificial Profile

We used a simple convergence technique, similar to `PAYL`, to stop the training of the artificial profile. At every certain interval (convergence check interval) we check if the $Manhattan$ [1] distance between the artificial profiles at the last interval and the current



(a) 1-gram (b) 2-gram

Figure 19: Anomaly score of Artificial Profile

interval is smaller than a certain threshold (convergence threshold). It stops training if the distance is smaller than the threshold. We set the convergence threshold (= 0.05) to be the same as the original implementation of `PAYL`. The artificial profile does not have to become very stable or match the normal profile perfectly because some deviation from the normal profile can be tolerated. To reduce the training time we set the convergence check interval to 2 packets. Thus, if we see two consecutive packets of a given length that are close to the learned profile, we stop training. Table 6 shows the number of packets required to converge the artificial profile of different packet lengths. As expected, the artificial profile converges very fast. The 1-gram profile converges faster than the 2-gram profile for the same packet length. We show that a small number of packets are enough to create an effective polymorphic blending attack. In practice, the attacker can use more learning data to create a better profile.

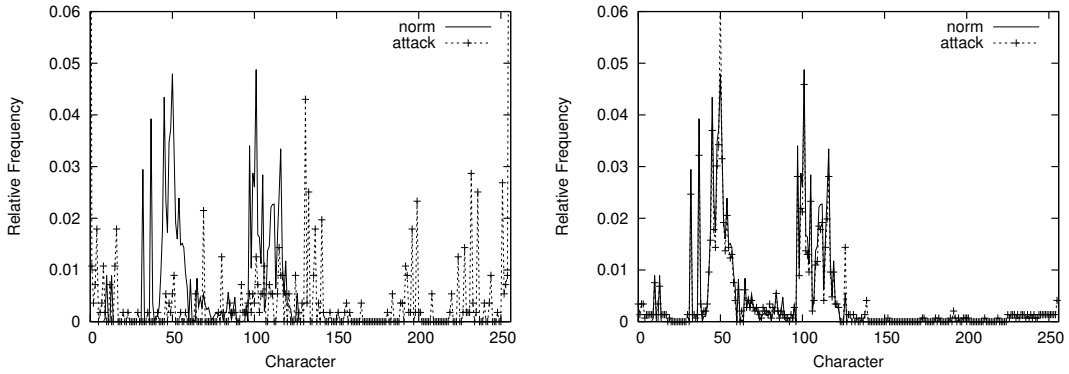
Figure 19 shows the anomaly score of the artificial normal profile, as calculated by the IDS normal profile, versus the number of attack training packets used to learn the artificial profile. As the number of attack training packets increases, the anomaly score of artificial normal profile decreases, which means that the artificial profile trained using more packets is a better estimation of the `PAYL` normal profile. The score needs to be less than the anomaly threshold of `PAYL` for the blending attack packets to have a realistic chance of evading `PAYL`. For all attack training sizes shown

in Figure 19, the score is well under the threshold (Table 5) used to configure PAYL to detect all the traditional (without blending) polymorphic attack instances.

4.2.7.2 Blending Attacks for 1-gram and 2-gram PAYL

For each packet length, we generated both the 1-gram and 2-gram PAYL normal profiles using the entire IDS training dataset (i.e., the first 14 days of HTTP traffic). For each packet length, the 1-gram and 2-gram artificial normal models were learned using a fraction of the attack training dataset. The learning stops at the point the models converge, as shown in Table 6.

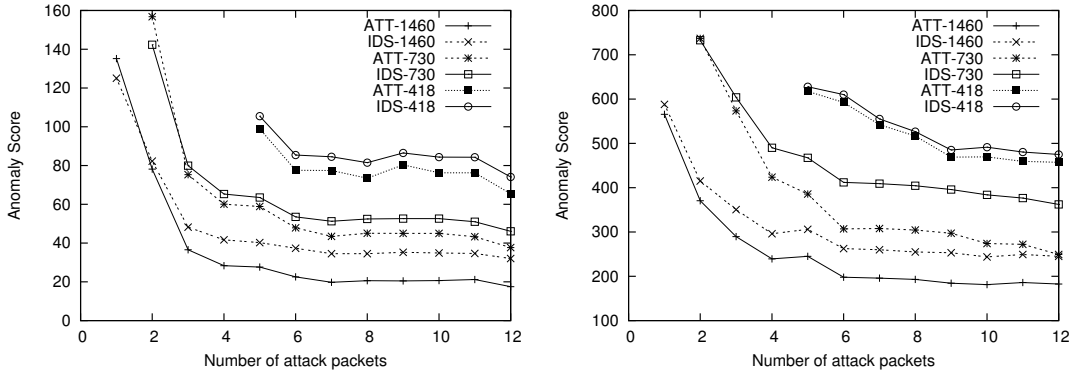
We used the one-to-one single-byte substitution technique discussed in Section 4.2.3.2.1 for constructing the blending attack against 1-gram PAYL, and the single byte encoding scheme discussed in Section 4.2.4.2 for the blending attack against 2-gram PAYL. Two sets of blending experiments were performed. In the first set of experiments, the substituted attack body was divided into multiple packets and each packet was padded separately to match the normal profile. A single decoding table is required to decode the whole attack flow. In the second set of experiments, the attack body was first divided into a given number of packets. Each of the attack body sections were substituted using one-to-one single byte substitution and then padded to match the normal frequency. Individually substituting the attack body for each packet allowed us to match the statistical profile of the substituted attack body closer to the normal profile. But it requires a separate decoding table for each packet, thus reducing the padding space considerably. For convenience, we call the first set of experiments *global substitution*, and the second *local substitution*. If $m > n$ for any of the above experiments, we simply substituted the low frequency attack characters using non-existing characters in the normal. This increased the error in blending attack but reduced the complexity of the blending attack algorithm. Figure 20 shows the comparison of the frequency distribution of different characters



(a) Original attack packet (b) 1-gram Blending Packet for packet length 1460

Figure 20: Comparison of frequency distribution of normal profile and attack packet

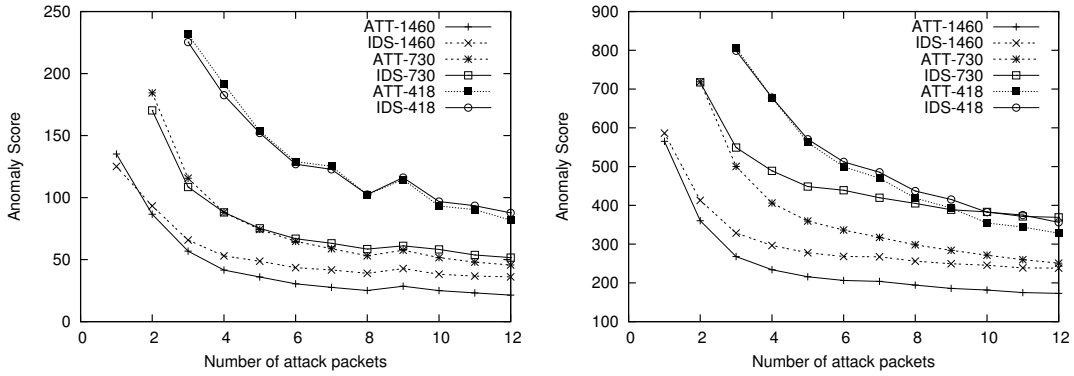
present in the HTTP traffic. The byte frequency distribution of the original attack instance is very different from the normal profile because the normal data has mainly printable ASCII characters whereas the attack payload has many characters that are unprintable. Thus, this was easily detected by both 1-gram and 2-gram IDS models. The attack was substituted and padded to obtain a single packet of length 1460. As shown in Figure 20(b), the frequency distribution of attack payload after substitution and padding becomes almost identical to the `PAYL` normal profile. This demonstrates the effectiveness of our polymorphic blending techniques. We studied how dividing



(a) 1-gram (b) 2-gram

Figure 21: Anomaly score of the blending attack packets (with local substitution) for artificial profile and IDS profile

an attack instance into several packets and blending them separately help match



(a) 1-gram (b) 2-gram

Figure 22: Anomaly score of the blending attack packets (with global substitution) for artificial profile and IDS profile

the attack packets with the artificial profile and evade `PAYL`. The experiments were performed with the number of attack packets ranging from 1 to 12. We checked the anomaly score of each attack packet as calculated by both the artificial profile and the IDS profile. Similar to the anomaly score of attack instances generated by CLET, the anomaly score of a blending attack instance was calculated as the highest of all the scores obtained by the attack packets corresponding to the blending attack instance. Figure 21 and Figure 22 show the anomaly scores of blending attacks with local substitution and global substitution, respectively. For each attack flow, we show the score of the packet with the highest score. It is evident that if the attack is divided into more packets, it matches the profile more closely. The reason is that if the attack body is divided into multiple fragments, for each packet there is more padding space available to match the profile. Also, local substitution works better than global substitution scheme for all cases except for 2-gram blending for packet length 418. Since our substitution table contains only normal 1-grams but may contain foreign 2-grams, a large substitution table may produce a large error for the 2-gram model. Considering that small packets have small padding space to reduce the error caused by the substitution table, having an individual substitution table in each packet can cause large error.

Although the score of the blending attack as calculated by the IDS model is greater than the score calculated by the artificial normal profile, it is still much lower than the anomaly threshold set for the detection of traditional polymorphic attacks.

Thus, our experiment clearly shows that unlike traditional polymorphic attacks, our blending attack is very effective in evading 1-gram and 2-gram `PAYL` for all the packet lengths and number of attack packets.

Table 7: Anomaly thresholds for different false positive rates in IDS models. Bracketed entries are the the numbers of packets required to evade the IDS using the local and global substitution scheme, respectively.

False Positive	418		730		1460	
	1-gram	2-gram	1-gram	2-gram	1-gram	2-gram
0.1	61.07 (17,-)	373.4 (-,12)	63.70 (5,7)	467.6 (5,5)	74.50 (3,3)	447.7 (2,2)
0.01	78.61 (12,15)	456.9 (22,8)	143.6 (2,3)	625.5 (3,3)	81.98 (3,3)	531.0 (2,2)
0.001	125.5 (5,7)	561.8 (7,6)	164.6 (2,3)	670.5 (3,3)	239.2 (1,1)	931.9 (1,1)
0.0001	166.8 (5,5)	582.6 (7,5)	244.5 (2,2)	805.0 (2,2)	243.4 (1,1)	935.0 (1,1)

4.2.7.3 Impact of IDS False Positive Rate

We also studied the effect of false positive rates on the detection of blending attacks. Anomaly threshold for a given false positive rate (fp) is set such that only fp fraction of normal data has anomaly score higher than the anomaly threshold. The anomaly thresholds for different false positive rates are shown in Table 4.2.7.2. The number of attack packets required to evade the IDS successfully for a given threshold is shown in the parenthesis. As we increase the false positive rate, we need to divide the attack into more packets to keep the score below the anomaly threshold. Thus, keeping a high false positive rate may increase the size of the blending attack. From the table we can infer that even if the IDS keeps its false positive rate high to detect more attacks, blending attack can still easily evade it using an attack size as small as 3,650, i.e. five packets of length 730.

Since 2-gram PAYL records some sequence information along with byte frequencies, it seems to be a good representation of normal traffic. In our experiments we found that 2-gram PAYL consistently produces higher anomaly score than 1-gram PAYL for all attack packet lengths. But at the same time, the 2-gram IDS needs to set very high anomaly thresholds to avoid high false positive rates. Thus, in practice, the 2-gram PAYL is actually only marginally more effective than the 1-gram version in detecting attacks.

Blending attacks can be successfully launched on both 1-gram and 2-gram models. Larger packet lengths are more suitable for blending attacks. With few exceptions, the local substitution scheme works better than the global substitution scheme. The 2-gram model provides only marginal advantage over the 1-gram model in detecting blending attacks but requires huge space to store the model. Thus, the 2-gram model may not be a better choice over the 1-gram model.

4.3 A Formal Framework

In Section 4.1 & 4.2, we presented basic concepts behind polymorphic blending attacks. The principal observation is that a network IDS, monitoring high-speed and high-volume traffic, typically uses simple statistical features instead of complex structural or semantic information to model the normal traffic. An attacker can exploit this simplicity or limitation to devise attacks capable of evading the IDS. However, the blending techniques presented earlier are not general enough. They are based on some heuristics which work well for PAYL but not necessarily other anomaly detection systems. We would like to develop techniques which can be used to generate PBAs for a wide range on anomaly IDSs.

In this section, we study the following problem: given an anomaly detection system and an attack, can one automatically generate PBA instances? Our approach is to develop a formal framework that starts with the models for IDSs and different

sections of a polymorphic attack. Based on these models, we can then reason about the complexity of the problem of generating a PBA, and develop general algorithms for solving the problem. We first discuss the class of IDS targeted by polymorphic blending attacks present in this section. We also present models for different attack sections. Then we discuss the structure of the polymorphic blending attack and detailed steps used to generate polymorphic blending attack. Based on the models, we reason the complexity of matching different attack sections with the normal model.

4.3.1 Modeling Anomaly Detection Systems

Earlier, we considered a class of anomaly detection systems that use only simple byte statistics of the normal traffic. We would like to generalize the concept of polymorphic blending attack to include a wide range of anomaly detection systems that use other structural information of the normal traffic.

Since a polymorphic attack typically mutates only the packet payload, we limit our scope to payload-based anomaly detection systems. These systems record the statistics and structure of the bytes preset in the normal network traffic packets. Such anomaly IDS proposed by researchers include PAYL [71], NETAD[35], LERAD [36], service-specific IDS [30], and structure based detection of Web attacks by Kruegel et al. [31]. We observe that these IDS can be represented as either Finite State Automaton (FSA) or equivalently stochastic Finite State Automaton (sFSA). sFSA is similar to FSA and has a probability assigned to all the transitions in the FSA.

4.3.1.1 PAYL

PAYL records the average frequency of different unique n-grams that appear in normal traffic packets. An n-gram model can be described using an sFSA where each state represents the unique $(n-1)$-gram corresponding to the last $(n-1)$ bytes in the packet. A transition from state $A(a_0a_1\cdots a_{n-2})$ to state $A'(a_1a_2\cdots a_{n-1})$ exists if and only if n-gram $(a_0a_1....a_{n-2}a_{n-1})$ is present in the normal traffic. The probability

of a transition is equal to the probability of the corresponding n-gram in the normal traffic. Every state is a start state and every state is an accept/end state. For example, 1-gram model can be represented using a single state sFSA: for every unique byte in the normal traffic, there exists a transition from the state to itself; and the probability of the transition is the same as the frequency of the byte in normal traffic.

4.3.1.2 Anagram

Anagram [70] is an another payload based network anomaly detector based on n-gram matching. Anagram records all the n-grams which are commonly present in normal traffic. However unlike PAYL, Anagram does not record the frequency of n-gram. Also, Anagram uses higher value of n ($4 \leq n \leq 7$) than PAYL ($n = 1 or 2$). An FSA can be used to represent all the n-grams stored by Anagram. Each state represents the unique $(n-1)$-gram present in the packet. A transition from state $A(a_0 a_1 \cdots a_{n-2})$ to state $A'(a_1 a_2 \cdots a_{n-1})$ exists if and only if n-gram $(a_0 a_1 .... a_{n-2} a_{n-1})$is present in the Anagram model.

4.3.1.3 NETAD and LERAD

Mahoney et al. presented a series of anomaly IDSs that use some network level data along with some payload data to detect intrusions. These systems use attributes such as bytes or words present at specific positions in the payload. An application layer LERAD rule is of the form (if, $word_1 = x_1, \cdots, word_{m-1} = x_{m-1}$ then $word_m \in \{x_{1,m}, \cdots, x_{n,m}\}$). Such a rule can be seen as regular grammar of the form $(x_1\{spaces\}x_2 \cdots x_{m-1}\{spaces\}\{x_{1,m}| \cdots |x_{n,m}\})$. Multiple rules can be combined using the '&' term to obtain a single regular grammar.

4.3.1.4 Structure-Based Systems

Kruegel et al. presented an IDS for Web services. A Web traffic packet is divided into attributes and different attributes are recorded using different byte characteristics,

including: attribute length, byte frequency, byte structure using sFSA, and token set. As in PAYL, byte frequency can be represented using a sFSA with one state. Token set ($T = \{t_1, \cdots, t_n\}$) can be seen as a regular grammar of the form $(t_1|\cdots|t_n)$. Models of the different attributes can be combined to form a single sFSA. The length constraint on an attribute is handled separately during the blending attack generation.

In summary, the above anomaly detection systems can be represented using a (s)FSA. For convenience, the (s)FSA corresponding to an IDS is called $(s)FSA_{ids}$.

4.3.1.5 Distance Calculation

Along with an (s)FSA model, an anomaly detection system uses a classifier to determine if an observed packet matches the (s)FSA or not. First the IDS finds a path taken by the monitored packet in the normal (s)FSA. For an FSA IDS, we calculate the number of times each error transition is covered by the monitored packet. For an sFSA IDS, we calculate the frequency with which each transition is taken in the sFSA. The frequency of each transition in the path is then used as a feature by the classifier to determine the distance between the monitored packet and the normal profile.

This distance metric should be small for packets that are accepted by the FSA with few errors and vice versa. In case of sFSA, the distance should be small if for all the transitions, the number of times the transition is taken in the monitored packet is proportional to the transition probability. If the distance is smaller than a threshold, the packet is considered normal. Otherwise, the packet is considered anomalous.

The IDSs discussed above use a simple distance metric for the classifier. For an FSA IDS, the distance metric is defined using a weighted sum of all the error transitions taken by the monitored data.

There are several distance metrics that are used for sFSA IDS. A typical IDS distance metric considered in this framework is weighted L_1 distance metric. The

distance between an $sFSA_{ids}$ and a monitored packet is defined as in equation 16. However, the results presented for the framework should be applicable to other distance metrics.

$$d = \sum_{t} w_t \times |p_t - \frac{l_t}{l}|, \tag{16}$$

where, p_t is the probability of transition t, l_t is the number of times transition t is taken in the monitored packet and l is the length of the path taken by the monitored packet.

4.3.1.6 An Example

We have shown that a wide range of intrusion detection systems that use byte statistics and structure of a normal packet can be represented using either an FSA or an sFSA. One main reason for using an FSA (or equivalently, regular expression) is that determining whether a string is generated by an FSA is very fast, and thus the IDS can be used to monitor high speed networks.

In this work, we assume that the attacker is trying to evade an IDS that can be represented using either an FSA or sFSA. Figure 23 shows a simple example of a sFSA IDS. The IDS accepts strings containing only following tuples: *ab*, *ba*, and *bb*. We use this simple IDS as a running example throughout this section.

The distance of a string from FSA is defined by Equation 17.

$$dist = \sum_{c_1, c_2 \in \{a,b\}} |\frac{n_{c_1c_2}}{l-1} - p_{c_1c_2}| \tag{17}$$

where $n_{c_1c_2}$ denotes the number of times transition c_1 to c_2 is taken by the input string, l is the length of the string, and $p_{c_1c_2}$ is the probability of transition c_1 to c_2 in the sFSA.

4.3.2 Polymorphic Attacks Section Models

Now we model different attack sections of a polymorphic attack: Attack Vector; Polymorphic Decryptor; Encrypted Attack Code; Key; and Padding.

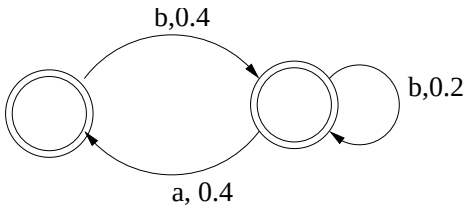


Figure 23: Simple sFSA IDS containing 3 tuples

$$\underbrace{aaba}_{\text{AV}}\ \underbrace{cbb}_{\text{Dec}}\ \underbrace{k_1k_2}_{\text{Key}}\ \underbrace{00h\ 01h\ 00h\ 00h}_{\text{Attack Code}}$$

$$\underbrace{aaba}_{\text{AV}}\ \underbrace{cbb}_{\text{Dec}}\ \underbrace{k_1k_2}_{\text{Key}}\ \underbrace{00h \oplus k_1\ 01h \oplus k_2\ 00h \oplus k_1\ 00h \oplus k_2}_{\text{Encrypted Attack Code}}$$

Figure 24: Simple attack example. Attack code is 4 byte string with `NUL` and `SOH` `ASCII` characters.

4.3.2.1 Attack Vector

Multiple techniques exist to mutate an attack vector. These mutation techniques modify the network protocol data along with the application layer data, while ensuring that the modified attack instance will still compromise the system. Some of the common techniques are multiple protocol rounds, divergence from protocol specification, `SSL` null record insertion, `TCP` fragmentation, `HTTP` padding, and shellcode polymorphism.

Regular grammars or a state-based machines have been commonly used by the developers of misuse detection systems to write attack signatures. Several of these signatures are used to detect malicious payload. These signatures primarily represent parts of attack vectors. Snort uses a regular expression to define an attack. NETSTAT uses a state machine to represent attack events. Several other misuse IDSs uses an equivalent of regular expression for attack specification.

Recently, Shai et al proposed a grammar based approach (GARD) to represent all the possible mutations of an attack. The attack was divided into three phase: attack pre-condition, attack execution, and attack confirmation. Each of these phases were

represented using a state automaton. Also, in their previous work (AGENT) Shai et al modeled attacks using a regular grammar generated using a set of inference rules.

In this framework, we consider attack vectors that can be represented using an FSA. For convenience, the FSA corresponding to an attack vector is called FSA_{av}.

4.3.2.2 Polymorphic Decryptor

Some of the shell-code obfuscation techniques used by attackers to generate polymorphic decryptors are:

- Garbage insertion: Operations which does not have any effect on the semantics of the decryptor. For example, modifying unused registers and memory locations.
- Instruction reordering: Re-ordering instructions that do not depend on each other.
- Equivalent code substitution: Substituting an instruction or a set of instructions with another set of instructions with same functionality.
- Register shuffling: Using a different register to store a given variable.

Most of the shell code obfuscation techniques are context-aware operations and thus the set of all obfuscated code cannot be represented sufficiently using context-free languages. Thus, we need a context-sensitive language to represent polymorphic decryptor code.

4.3.2.3 Attack Code

Similar to polymorphic decryptor, attack code can also be mutated using different shell-code obfuscation techniques presented in Section 4.3.2.2. Thus, the language representing attack code also needs to be context sensitive.

Attack Vector	Decryptor	Encrypted Attack Code	Key	Padding

Figure 25: Position of different attack section in attack.

4.3.2.4 Key

Attacker can theoretically encrypt an attack code using any encryption key without affecting the working of the attack. Thus, the language representing a key is simply all the possible strings of length k, where k is the length of the key. In other words, key is Σ^k, where Σ is the set of all characters.

4.3.2.5 Padding

Similar to key, an attacker can pad an attack packet with any random data. Therefore, the language representing the padding is $Sigma^{l_p}$, where l_p is the length of the padding.

4.3.3 Polymorphic Blending Attack

As discussed earliler, a polymorphic blending attack contains up to five sections. An attack vector is required to be present in all the attacks. An attack code is required if adversary wants to execute an arbitrary code at victim. A polymorphic decryptor and a decryption table need to be present if adversary decides to encrypt the shell-code. Padding is optional and may help adversary in closely matching the normal profile.

The positioning of each attack section in an attack may depend on the semantics of the attack, which is decided by the adversary when designing the attack. In this discussion, we assume that all the five sections are present in the attack packet and they are positioned as showed in Figure 25. However, the discussion on generating polymorphic blending attack and its complexity should hold for other positioning of the sections. The discussion also applies even if some of the attack sections are missing.

4.3.3.1 Polymorphic Blending Attack Steps

Lets assume that an attacker has learned the (s)FSA corresponding to the (artificial) normal profile of the targeted IDS. The next step is to design an attack packet that can be accepted by the (s)FSA.

To generate a polymorphic blending attack, adversary needs to generate different attack sections that matches the normal profile. First adversary decides the positioning of each attack section in the attack. Then each attack section is generated in a separate step. Each section is matched to the normal profile in the same order as the section is positioned in the attack. We use the positioning shown in Figure 25 for our polymorphic blending attack.

First we generate a suitable attack vector. The choice of attack vector depends on the vulnerability on the victim side. To stay undetected, attack vector should be small and should also look like normal. For an FSA IDS, the attack vector should traverse the FSA with minimal number of invalid transitions. For an sFSA IDS, in addition being taking valid transitions, the attack vector should also match the transition probabilities.

Then we adjust the (s)FSA for what have been already matched by the attack vector. First, we identify the path taken by the attack vector in the (s)FSA. If there does not exist such a complete path (e.g., some transitions are not in the (s)FSA), there will be an error matching the attack vector with the normal profile. The start state of the (s)FSA is set to the end state of the path traversed by the attack vector. For an sFSA IDS, the transition probabilities of each transition is also adjusted according to the number of times the transition was taken in the attack vector.

The next step is to generate a polymorphic decryptor. Similar to the attack vector, the polymorphic decryptor should also match the adjusted (s)FSA. After generating the polymorphic decryptor, the start state of the (s)FSA is adjusted for the generated

decryptor. In case of an sFSA, the transition probabilities of the sFSA is also adjusted.

The next step is to generate an attack code. The choice of the attack code depends on what adversary wants to do after she has compromised the system. After deciding on the attack code, adversary needs to determine an encryption key to encrypt the attack code. The requirement on the encryption key is that the encrypted attack code and the decryption key should be accepted by the adjusted (s)FSA (i.e., there exists a corresponding path). For a sFSA, an additional requirement is to also match the transitions probabilities.

The final step is to pad the attack packet to get a desired packet length. For an sFSA IDS, padding can be used to match the final attack packet even closer to the normal profile.

We use a simple blending attack (shown in Figure 24) to demonstrate different concepts presented in the paper. We use an *XOR* encryption scheme with key length 2 (obviously, the encryption key and the decryption key is the same in an *XOR* scheme). The attack vector, decryptor, key, and attack code are concatenated in the given order to produce an attack packet payload.

For convenience of discussion, we denote the string corresponding to the decryption key concatenated with the encrypted attack code as S_{key_ac}.

4.4 Formal Analysis

In this section, we discuss the complexity of matching different attack sections with the normal model. We also present algorithms that can used to match each attack sections with the normal model.

4.4.1 Attack Vector

Our aim is to generate an attack vector that matches the normal profile. We consider attack vectors that can be represented using an FSA. We show that for an FSA IDS, it is easy to generate an attack vector of given length that is accept by the FSA with

minimum number of errors. But generating an attack vector with minimum distance to the sFSA IDS is NP-complete problem.

4.4.1.1 FSA IDS

To find a matching attack vector, we need to find a string which is accepted by both FSA_{av} and FSA_{ids}. We find the intersection of FSA_{av} and FSA_{ids}, which will also be an FSA. If the intersection is not empty, adversary can obtain a desired attack vector by generating a string that is accepted by the intersected FSA. In case the intersection is empty, following algorithm can be used to find an attack vector string that is accepted by the IDS with minimum errors.

First, we add all the error states and transitions in the IDS FSA. For all the valid transitions, the cost of the transition is set to 0. For all the error transitions, the cost of the transition is set to 1. We call this IDS with error transitions and cost, $augFSA_{ids}$. Then we compute the intersection of FSA_{av} and $augFSA_{ids}$. We the intersection FSA, $FSA_{av \cap augids}$. To generate an attack vector of given length l_{av}, we need to find a minimum cost path of length l_{av} from the start state q_0 to an end state in $FSA_{av \cap augids}$. This problem is very similar to shortest path problem and can be solved using a polynomial-time algorithm. The algorithm is very similar to Bellman-Ford algorithm which is used to solve single source shortest path problem.

We maintain two arrays, d_q and p_q of length l_{av} at each state q of $FSA_{av \cap augids}$. $d_q[i]$ at node q contains the minimum distance from q_0 to q using path of length i. $p_q[i]$ contains the previous node in that path. The algorithm contains of l_{av} steps. In the ith step, algorithm determines the min-cost path of length i from q_0 to q, $\forall q \in Q$.

The min-cost path of length i is determined by relaxing every transition in $FSA_{av \cap augids}$. An edge (q_1, q_2) is relaxed by testing whether the min-cost distance to state q_2 can be reduced by taking the ith transition of the path through q_1. If the min-cost distance is reduced, the $d_{q_2}[i]$ is updated as $d_{q_1}[i-1] + cost(q_1, q_2)$. The pseudo-code of the

algorithm is given in Appendix B.5.

Once we find a min-cost path of length l_{av} from start state to all the other states, we choose the final state which has minimum cost path of length l_{av}.

If an adversary wants to generate min-error attack vector, irrespective of the length of the attack vector, we can use Bellman-Ford algorithm to find min-cost path from q_0 to all the final states. Again, we choose the final state which has minimum cost path.

4.4.1.2 sFSA IDS

In the case of sFSA IDS, along with matching the transitions, the attack vector should also match the transition probabilities. The distance between the sFSA and attack vector is calculated using the distance metric in equation 16. We show that unlike matching FSA IDS, matching attack vector to sFSA IDS is a hard problem. It is impractical to try brute force by generating all the possible attack vectors and try matching them to the sFSA. The brute force approach will take time exponential to the number of nodes in the FSA_{av}. We prove that generating attack vector that matches the sFSA within a given error bound is NP-complete.

We formally define the problem of finding an attack vector that is accepted by the sFSA IDS.

Definition 4.4.1 *Given an attack vector FSA_{av} and an IDS $sFSA_{ids}$, the problem PBA_{av}^{sFSA} is to find a string which is accepted by FSA_{av} and also matches the transition probabilities of $sFSA_{ids}$ within an error bound, ϵ.*

Theorem 4.4.1 *Problem PBA_{av}^{sFSA} is NP-complete.*

Proof For a problem to be NP-complete, the problem should be in NP and should be NP-hard.

A problem is in NP if a given solution can be verified for its correctness in polynomial time. Given an attack vector string, we can easily verify in polynomial time

whether or not the string is accepted by the FSA_{av}. We can also verify if the distance between the attack vector and the $sFSA_{ids}$ is within the error bound. Thus, we can efficiently verify if the string is a correct solution or not.

To prove that the problem is NP-hard, we reduce a well known NP-complete problem, namely Hamiltonian cycle problem, to PBA_{av}^{sFSA}. For a given directed graph $G(V, E)$, we create an FSA_{av} and $sFSA_{ids}$ such that solution of PBA_{av}^{sFSA} is equivalent to the Hamiltonian cycle in graph $G(V, E)$.

Let Σ be the set of input characters which contains a character c_i for each vertex $v_i \in V$. Suppose w is an arbitrarily big positive integer which is multiple order of magnitudes greater than $\|V\|\|E\|$. The FSA_{av} is the set of all the substrings of length $2\|V\|$, that is, $FSA_{av} = \Sigma^{2\|V\|}$.

The $sFSA_{ids}$ is constructed as follows. For each vertex $v_i \in V$, we create two states q_{i_1} and q_{i_2}. We introduce a transition from state q_{i_1} to state q_{i_2} on input c_i with transition probability of $\frac{1}{2\|V\|}$ and cost w. We denote this kind of transition as T_v. Also, for each edge (v_i, v_j) in E, we introduce a transition of type T_e from state q_{i_2} to state q_{j_1} on input c_j with transition probability $\frac{1}{2\|E\|}$ and cost 1. States $q_{i_1}, 1 \leq i \leq \|V\|$ are both start state and final state. Figure 26 shows construction of a $sFSA_{ids}$ for a sample directed graph. The cost of taking a non-existing transition in the graph is w. Please note that any path in above $sFSA_{ids}$ which consists of only valid transition will be of form $T_vT_eT_vT_e \cdots T_vT_e$ and corresponding string will be of form $c_{i_1}c_{i_2}c_{i_2}c_{i_3}c_{i_3} \cdots c_{i_l}c_{i_l}c_{l+1}$.

Now, we prove that this transformation is indeed a reduction and graph G has a Hamiltonian cycle if and only if there exists an attack vector which matches the $sFSA_{ids}$ with an error bound of $\frac{\|E\|-\|V\|}{\|E\|}$.

Suppose, graph G has a Hamiltonian cycle $v_1v_2 \cdots v_{\|V\|}v_1$. Then we can construct an attack vector $c_1c_2c_2c_3c_3 \cdots c_{\|V\|}c_{\|V\|}c_1$. This attack vector takes only the valid transitions in $sFSA_{ids}$. The transition probabilities are matched exactly for

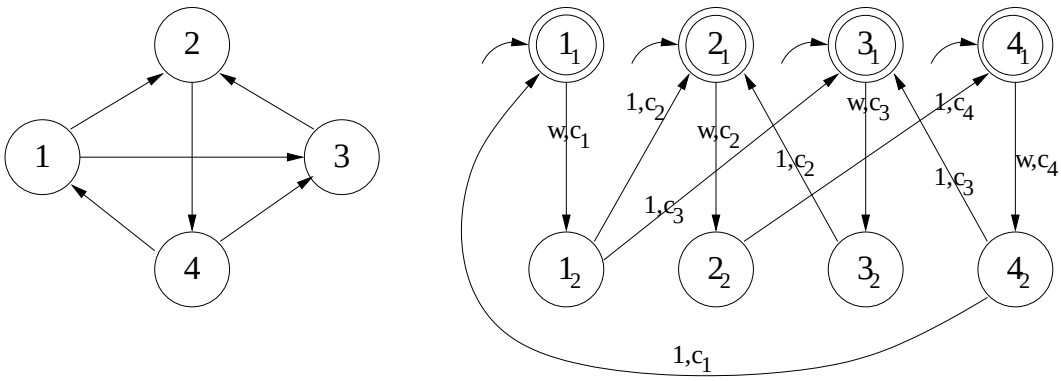


Figure 26: Construction of a $sFSA_{ids}$ for a given Hamiltonian graph

all type T_v transitions. Thus error contributed by such transitions is zero. For all type T_e transitions (q_{i_2}, q_{j_1}) traversed by the attack vector, the error contributed is given in equation 18. There are $\|V\|$ such T_e transitions covered by the attack vector. Therefore, total error due to such transitions is $\frac{\|E\|-\|V\|}{2\|E\|}$. For all the valid transitions (q_{i_2}, q_{j_1}) not traversed by the attack vector, the error contributed is given in equation 19. There are $(\|E\| - \|V\|)$ such T_e transitions not covered by the attack vector and the total error due to such transitions is $\frac{\|E\|-\|V\|}{2\|E\|}$. Thus, the final error is $\frac{\|E\|-\|V\|}{\|E\|}$ which is precisely the bound.

$$err_{ij} = 1 * (\frac{1}{2\|E\|} - \frac{1}{2\|V\|}) = \frac{\|E\| - \|V\|}{2\|E\|\|V\|} \tag{18}$$

$$err_{ij} = \frac{1}{2\|E\|} \tag{19}$$

Conversely, suppose there exists an attack vector of length $2\|V\|$ with error below $\frac{\|E\|-\|V\|}{\|E\|}$. Since this bound is smaller than w, attack vector takes only valid transitions in the $sFSA_{ids}$ and also takes all the transitions of type T_v exactly once. Thus, attack vector is of form $c_{i_1}c_{i_2}c_{i_2}c_{i_3}c_{i_3}\cdots c_{i_{\|V\|}}c_{i_{\|V\|}}c_{i_1}$. In this case, there exists an edge from each v_{i_j} to $v_{i_{j+1}}, 1 \leq j \leq \|V\|$. Thus, cycle $v_{i_1}v_{i_2}v_{i_3}\cdots v_{i_{\|V\|}}v_{i_1}$ is a Hamiltonian cycle. ∎

4.4.1.3 Approximate Solution

In section 4.4.1.2, we showed that the problem of finding an attack vector that matches the $sFSA_{ids}$ is NP-complete. That is, it may take time exponential to the desired length of the attack vector. Although an attacker may not have any time restrictions, a polynomial time solution is clearly more desirable. Rather than finding the optimal solution which tries to match the attack vector as close as possible, an attacker may simply apply some heuristics that produce an approximate solution using much less resources (time and memory). We can also reduce the PBA_{av}^{sFSA} problem to some other domain which has good approximate solvers.

4.4.1.3.1 Greedy Heuristic Greedy algorithms are used widely for finding approximate solutions. The idea is to make a local optimal decision at each stage. Using this we hope to get near the global optimal solution. The local decision for PBA_{av}^{sFSA} problem is made based on minimizing the current error. The greedy algorithm takes l_{av} stage and at each stage it generates a character in the attack vector.

We start at the start state of both FSA_{av} and $sFSA_{ids}$. We consider all the transitions from the start state in FSA_{av} that can lead to a final state in total l_{av} steps. Out of all these transitions we choose one that minimizes the distance between current generated attack vector string (which is simply the character corresponding to that transition in FSA_{av}) and $sFSA_{ids}$. Suppose after $i-1$ steps, the algorithm has already generated an attack vector string $c_1c_2\cdots c_{i-1}$ and it is at state q_{i-1} in FSA_{av}. At ith step, we consider all the transitions from q_{i-1} that can lead to final state in next $l_{av}-i$ steps. We choose the transition to state q_i that produces c_i such that, $c_1c_2\cdots c_{i-1}c_i$ is at the minimum distance to $sFSA_{ids}$.

4.4.1.3.2 Reduction A greedy heuristic may not give us a good approximate solution in many cases because the problem does not exhibit a clear optimal substructure. In such cases, we can reduce the PBA_{av}^{sFSA} problem to *Integer Linear Programming* (ILP) problem. ILP is known to be NP-complete, but there exist several good heuristics to solve big ILP problems. An ILP problem tries to find the minimum of a linear cost function over a set of variables. These variables are restricted by a finite number of linear constraints. All the variables in the solution should be integer.

First, we find an $FSA_{av \cap ids}$ that is an intersection of FSA_{av} and $sFSA_{ids}$. For every state ${q_i}^{av}$ and ${q_j}^{ids}$ in FSA_{av} and $sFSA_{ids}$ respectively, we have a state q_{ij} in $FSA_{av \cap ids}$. For every transition $t(i_1, i_2)^{av}$ and $t(j_1, j_2)^{ids}$ in FSA_{av} and $sFSA_{ids}$ respectively, we have a transition $t(i_i j_1, i_2 j_2)$ from state $q_{i_1 j_1}$ to state $q_{i_2 j_2}$ in $FSA_{av \cap ids}$. Now the PBA_{av}^{sFSA} problem is equivalent to finding a string that is accepted by $FSA_{av \cap ids}$ with minimum error as defined by equation 20.

$$d = \sum_{t(j_1,j_2) \in sFSA_{ids}} w_{t(j_1,j_2)} \times |\frac{\sum_{t(i_1,i_2) \in FSA_{av}} l_{t(i_1 j_1, i_2 j_2))}}{l_{av}} - p_{t(j_1,j_2)}| \tag{20}$$

Now we generate an ILP problem corresponding to a PBA_{av}^{sFSA} problem. For each transition $t(i_i j_1, i_2 j_2)$ in the $FSA_{av \cap ids}$, we have a variable $h_{i_i j_1, i_2 j_2}$ which represents the number of times transition $t(i_i j_1, i_2 j_2)$ is taken in the generated attack vector. The length of the path can be represented using equation 21.

$$\sum_{transition \ \ t \in FSA_{av \cap ids}} h_t = l_{av} \tag{21}$$

The necessary and sufficient condition for a valid path from start state to a final state can be represented using the following three equations.

$$\sum_{t \in IN(q_0)} h_t + err_{q_0} + 1 = \sum_{t \in OUT(q_0)} h_t, \tag{22}$$

$$\sum_{t \in IN(q_f)} h_t + err_{q_f} = \sum_{t \in OUT(q_f)} h_t + 1, \tag{23}$$

$$\sum_{t \in IN(q_{ij})} h_t + err_{q_{ij}} = \sum_{t \in OUT(q_{ij})} h_t, \qquad \forall q_{ij} \in FSA_{av \cap ids}, q_{ij} \notin \{q_0, q_f\} \tag{24}$$

where $IN(v)$ and $OUT(v)$ is the set of in and out transitions of vertex v, respectively. q_0 and q_f is the start state and final state, respectively.

The minimization criteria for the ILP problem which minimizes the distance between normal profile and attack vector can be written as:

$$\sum_{t(j_1,j_2) \in sFSA_{ids}} w_{t(j_1,j_2)} \times |\frac{\sum_{t(i_1,i_2) \in FSA_{av}} h_{t(i_1 j_1, i_2 j_2)}}{l_{av}} - p_{t(j_1,j_2)}| \tag{25}$$

Solving above ILP for the given minimization criteria provides the desired attack vector with minimum distance to the $sFSA_{ids}$.

4.4.2 Polymorphic Decryptor

A set of all polymorphic code that can be used to decrypt a given encrypted attack code can be represented satisfactorily using context sensitive language. Thus the problem of generating a suitable decryptor that matches the normal profile can be seen as a problem of finding an intersection of a context sensitive language and an (s)FSA. Problem related to finding an intersection between a context sensitive language and any other language in general is un-tractable. Thus, we cannot find an appropriate decryptor automatically. In this work, we assume that the decryption code is generated out of band. This decryption code should preferably be small and should contain mainly `ASCII` characters. Adversary can generate the decryptor either manually or using a polymorphic code engine. However, the adversary cannot guarantee that generate decryptor matches optimally to the (s)FSA IDS.

4.4.3 Padding

As discussed in section 4.3.2.5, padding can be represented using Σ^{l_p}, where l_p is the length of padding. Padding can be seen as a special case of attack vector where FSA for padding accepts all the strings of a given length. All the results regarding generating an attack vector can be applied to padding. A suitable padding for FSA IDS can be generated in polynomial time using the algorithm presented in section 4.4.1.1. NP-completeness proof, shown in section 4.4.1.2, can be used to prove that finding an optimal padding which matches a given sFSA IDS is NP-complete. Furthermore, the greedy algorithm and ILP reduction shown in section 4.4.1.3 can be used to find a suitable padding.

4.4.4 Encrypted Attack Code and Key

In section 4.2, we used a simple byte substitution scheme for encryption. During encryption, every attack character in the attack body is substituted by a normal character. To store the reverse substitution (or decoding) table of the simple byte substitution scheme, we use the same technique as before: the index of the decoding table determines the attack character, and the entry at an index is the normal character used to substitute the corresponding attack character.

The encryption techniques studied in this section include both *XOR* encryption scheme and the byte substitution scheme. It is important to consider *XOR* because there are several existing polymorphism tools that use *XOR* based encryption. These tools may be extended to generate PBA. Unlike substitution, the decryption key for *XOR* is the same as the encryption key, and can be stored in a straight forward manner.

Both substitution and *XOR* are very simple schemes and are used in more complex encryption schemes. By studying PBA with these simple schemes, we hope to develop a understanding as well as solutions applicable to more complex schemes.

We want to find an encryption key so that the attack packet sections of the decryption key and the encrypted attack code, or S_{key_ac}, can be accepted by the FSA or the adjusted sFSA (for convenience, in this section we simply call the adjusted sFSA a sFSA). We will show that this is a hard problem even when using very simple encryption schemes, namely, substitution and *XOR*. As a corollary, the problem is hard when using more complex encryption schemes. The most direct and important corollary, however, is that the problem of generating a suitable encrypted attack code is a hard problem.

A brute force approach to find the encryption key requires to generate every possible key and check the distance (as defined by the IDS) of the S_{key_ac} to the (s)FSA. For a simple substitution-based (encryption) scheme, this will take at least ${}^{n}P_{m}(n-m)^{n-m}$ iterations, where n is the number of unique normal characters and m is the number of unique attack characters. For *XOR* encryption with key of length l, finding an optimal polymorphic blending attack will take at least n^l iterations. These numbers can very large, and thus a brute force approach is often impractical.

In this section, we first analyze the hardness of finding an encryption key that ensures S_{key_ac} is a valid string accepted by the FSA_{ids} (without any transition probabilities). We show that this problem is NP-complete. Thus, it may not be solvable deterministically in polynomial time of the key length l or m. We show this result for both byte substitution based encryption and *XOR* based encryption. This result can be extended to show that even if we allow solution to contain ϵ fraction of invalid transitions, the problem is still NP-complete. We extend the above results and argue that the problem of finding an encryption key that optimally matches the S_{key_ac} to sFSA or finding a solution within an ϵ range of the optimal solution is also NP-complete.

4.4.4.1 Substitution Based Encryption Scheme

We formally define the problem of finding a substitution key that ensures S_{key_ac} is accepted by the FSA of an IDS.

Definition 4.4.2 *Given an attack code and the FSA of an IDS, the problem* PBA_{sub}^{FSA} *is to find a one-to-one mapping from attack characters to normal characters so that* S_{key_ac} *is accepted by the given FSA.*

Theorem 4.4.2 *Problem* PBA_{sub}^{FSA} *is NP-complete.*

Proof For a problem to be NP-complete, the problem should be in NP and should be NP-hard.

A problem is in NP if a given solution can be verified for its correctness in polynomial time. Given an one-to-one mapping, we can easily generate the decryption key (a table) and the encrypted attack code. Since FSA is a decidable language, we can verify in polynomial time whether or not S_{key_ac} string will be accepted by the FSA. Thus, we can efficiently verify if the one-to-one mapping is correct or not.

To prove that the problem is NP-hard, we reduce the well known 3-SAT problem to PBA_{sub}^{FSA}. Consider a 3-SAT problem with q, $q \leq 128$ variables and r clauses. If q is smaller than 128, we add dummy unused variables to make total number of variables 128. Suppose the 3-SAT problem is,

$$SAT = (x_{10} \vee x_{11} \vee x_{12}) \wedge (x_{20} \vee x_{21} \vee x_{22}) \wedge \cdots \wedge (x_{r0} \vee x_{r1} \vee x_{r2}),$$

where $x_{10}, x_{11}, \cdots, x_{r2} \in \{x_0, \overline{x_0}, x_1, \overline{x_1}, \cdots x_{127}, \overline{x_{127}}\}$.

Given the above 3-SAT, we design PBA_{sub}^{FSA} problem as follows. For every variable x_i in the 3-SAT, we have an attack character att_i, two normal characters $norm_i$ and $norm_{i+128}$, and a corresponding entry e_{att_i} in the decryption table. $e_{att_i}, \forall 128 \leq i \leq 255$, is a dummy decryption table entry. The value of the variable x_i is related to e_{att_i}

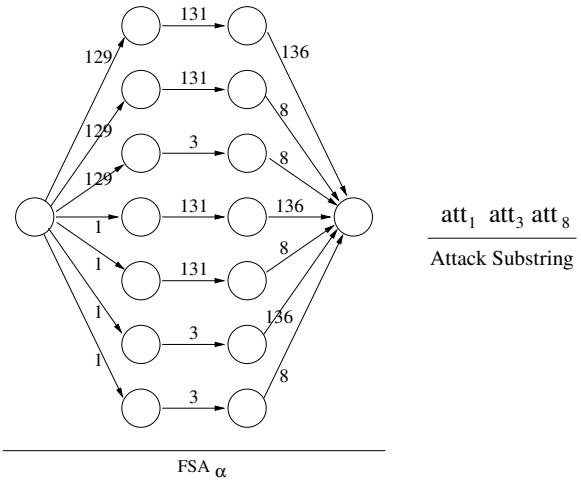


Figure 27: FSA_α and attack substring for clause $x_1 \vee \overline{x_3} \vee x_8$. For convenience, we represent $norm_i$ by just i.

as follows.

$$x_i = 1, \text{ if and only if } e_{att_i} = norm_i \text{ and } e_{att_{i+128}} = norm_{i+128}, \quad (26)$$

$$= 0, \text{ if and only if } e_{att_i} = norm_{i+128} \text{ and } e_{att_{i+128}} = norm_i \quad (27)$$

Table 8: Truth table and corresponding key table for clause $x_1 \vee \overline{x_3} \vee x_8$

x_1	x_3	x_8	e_{att_1}	e_{att_3}	e_{att_8}
0	0	0	$norm_{129}$	$norm_{131}$	$norm_{136}$
0	0	1	$norm_{129}$	$norm_{131}$	$norm_8$
0	1	1	$norm_{129}$	$norm_3$	$norm_8$
1	0	0	$norm_1$	$norm_{131}$	$norm_{136}$
1	0	1	$norm_1$	$norm_{131}$	$norm_8$
1	1	0	$norm_1$	$norm_3$	$norm_{136}$
1	1	1	$norm_1$	$norm_3$	$norm_8$

For every clause $clause_\alpha, 1 \leq \alpha \leq r$ in the 3-SAT, we construct a section of FSA (FSA_α) as shown in Figure 27. First, we construct the truth table of the clause (see Table 8). For each entry in the truth table, we have a path containing three transitions where each transition corresponds to the value of a variable in the truth table entry. In addition to $FSA_\alpha, 1 \leq \alpha \leq r$, we have a section of FSA (FSA_{KT}) of length 256 corresponding to the key.

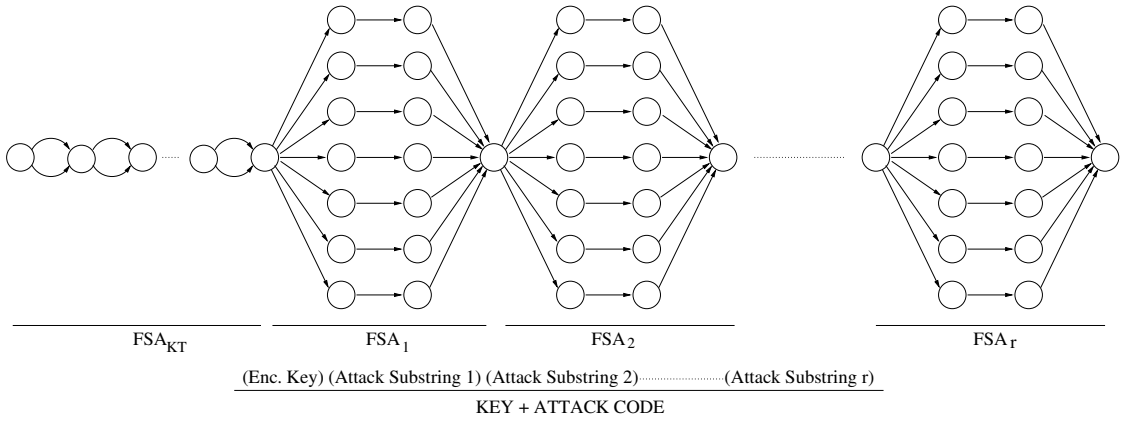


Figure 28: FSA and S_{key_ac} corresponding to the SAT problem

Also, for every variable x_i or $\overline{x_i}$ in a $clause_\alpha$, we have an attack character att_i in the attack code. Thus for every clause $clause_\alpha$, we have a substring (str_α) of length 3 in the attack code. Figure 27 shows an example attack code substring for a hypothetical clause. The encoded attack substring will be $e_{att_1} e_{att_3} e_{att_8}$.

In Figure 27, we can observe that the encrypted str_α is accepted by the FSA_α if and only if the encoding of attack characters are chosen from one of the entries in the given encoding table shown in Table 8. Since every entry in the encoding table corresponds to an entry in the truth table of the $clause_\alpha$, the encrypted str_α is accepted by the FSA_α if and only if $clause_\alpha$ is *true*.

The final FSA, FSA_{SAT}, and the attack code corresponding to the 3-SAT problem are shown in Figure 28. The construction of the above FSA_{SAT} takes polynomial time. Please note that there exists a solution to the given PBA_{sub}^{FSA} problem if and only if the encrypted str_α is accepted by FSA_α for all $1 \leq \alpha \leq r$.

If the above PBA_{sub}^{FSA} problem has a solution mapping $e_{att_i}, 0 \leq i \leq m-1$, then one can find assignments for variables $x_i, 0 \leq i \leq 127$ using Equation 27. Since S_{key_ac} is accepted by FSA_{SAT} for mapping $e_{att_i}, 0 \leq i \leq m-1$, the encrypted str_α is accepted by FSA_α for all $1 \leq \alpha \leq r$. However, the encrypted str_α is accepted by FSA_α only if $clause_\alpha$ is *true*. Thus, all clauses of the 3-SAT problem is *true* and the 3-SAT is satisfied.

Also, if there exists an assignment of variables x_i such that the 3-SAT problem is satisfied, then we can compute e_{att_i} using Equation 27. Since 3-SAT is satisfied, all $clause_\alpha$, $1 \leq \alpha \leq r$, are *true*. But $clause_\alpha$ is *true* only if the encrypted str_α is accepted by FSA_α. Thus, all encrypted str_α, $1 \leq \alpha \leq r$, are accepted by FSA_α, and S_{key_ac} is accepted by FSA_{SAT}.

From above, we can conclude that PBA_{sub}^{FSA} is at least as hard as 3-SAT. Since 3-SAT is NP-hard, PBA_{sub}^{FSA} is also NP-hard. Since PBA_{sub}^{FSA} is also in NP, PBA_{sub}^{FSA} is an NP-complete problem. ∎

4.4.4.2 XOR Encryption Scheme

We formally define the problem statement of finding a XOR encryption key that ensures S_{key_ac} is accepted by the FSA of an IDS.

Definition 4.4.3 *Given an attack code and the FSA of an IDS, the problem PBA_{xor}^{FSA} is to find an encryption key, of length l, so that S_{key_ac} is accepted by the given FSA.*

Theorem 4.4.3 *Problem PBA_{xor}^{FSA} is NP-complete.*

Proof The proof of NP-completeness of PBA_{xor}^{FSA} is similar to the proof of PBA_{sub}^{FSA}. First we show that PBA_{xor}^{FSA} is in NP. Given a solution encryption key $(k_0 k_2 \cdots k_{l-1})$, we can easily generate the encrypted attack code and hence the S_{key_ac}. We can easily verify in polynomial time if S_{key_ac} is accepted by the FSA or not. Thus, the problem is verifiable in polynomial time.

Now we show a reduction from 3-SAT to PBA_{xor}^{FSA}. Consider a 3-SAT problem with q, $q \leq 128$ variables and r clauses. If q is smaller than 128, we add dummy unused variables to make total number of variables 128. Suppose the 3-SAT problem is,

$$SAT = (x_{10} \vee x_{11} \vee x_{12}) \wedge (x_{20} \vee x_{21} \vee x_{22}) \wedge \cdots \wedge (x_{r0} \vee x_{r1} \vee x_{r2}),$$

where $x_{10}, x_{11}, \cdots, x_{r2} \in \{x_0, \overline{x_0}, x_1, \overline{x_1}, \cdots x_{127}, \overline{x_{127}}\}$.

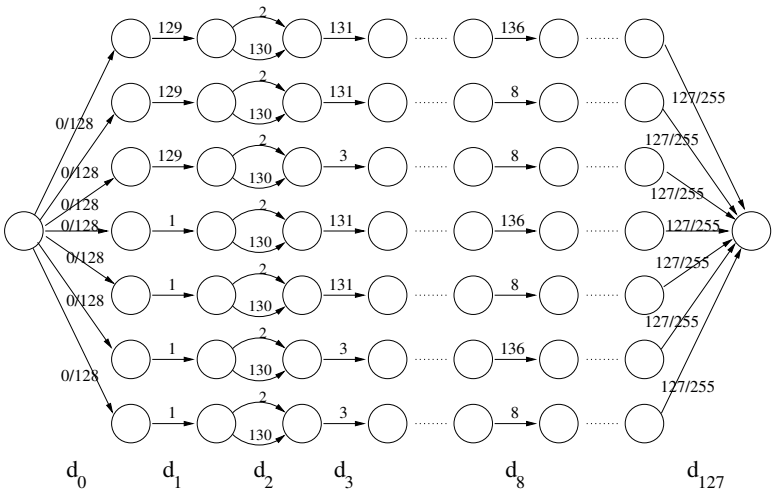


Figure 29: FSA_α for clause $x_1 \vee \overline{x_3} \vee x_8$

Given above 3-SAT, we design PBA^{FSA}_{xor} problem as follows. For every variable x_i in 3-SAT, we have an attack character att_i and two normal characters: $norm_i$ and $norm_{i+128}$. The value of the variable x_i is related to XOR encryption key k_i at ith position is as follows.

$$x_i = 1, \text{ if and only if } k_i = norm_i, \tag{28}$$

$$= 0, \text{ if and only if } k_i = norm_{i+128} \tag{29}$$

Table 9: Truth table and corresponding key table for clause $x_1 \vee \overline{x_3} \vee x_8$

x_1	x_3	x_8	k_1	k_3	k_8
0	0	0	$norm_{129}$	$norm_{131}$	$norm_{136}$
0	0	1	$norm_{129}$	$norm_{131}$	$norm_8$
0	1	1	$norm_{129}$	$norm_3$	$norm_8$
1	0	0	$norm_1$	$norm_{131}$	$norm_{136}$
1	0	1	$norm_1$	$norm_{131}$	$norm_8$
1	1	0	$norm_1$	$norm_3$	$norm_{136}$
1	1	1	$norm_1$	$norm_3$	$norm_8$

For every clause $clause_\alpha$ in the 3-SAT, we have a FSA section, FSA_α, which is generated as follows (see Figure 29). First, we construct the truth table for $clause_\alpha$ as shown in Table 9. For every entry in the truth table, we create a path of length 128 in FSA_α. If a variable x_i is present in the clause, its corresponding ith transition in

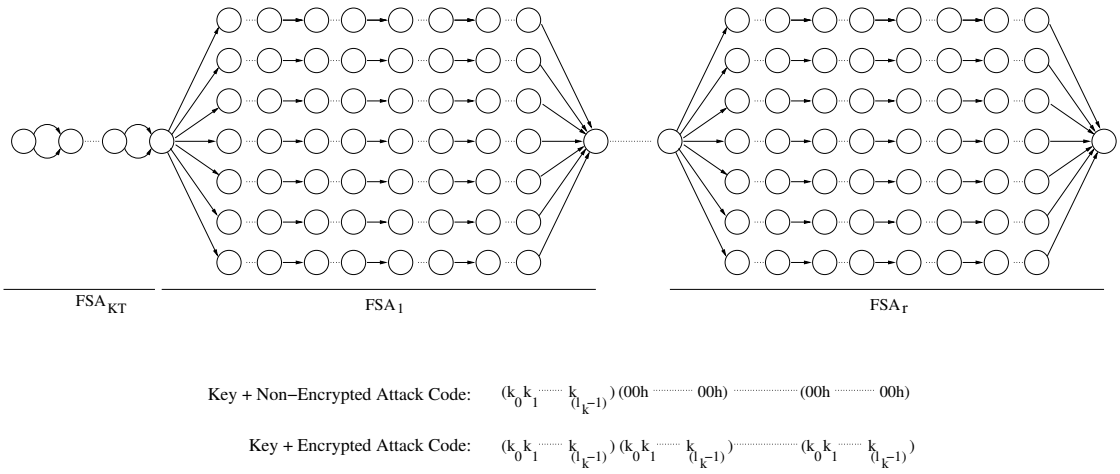


Figure 30: FSA and S_{key_ac} corresponding to the SAT problem

FSA_α has transition for $norm_i$. Otherwise, if variable $\overline{x_i}$ is present in the clause, its corresponding ith transition in FSA_α has transition for $norm_{i+128}$. If neither x_i nor $\overline{x_i}$, is present in the clause, then the corresponding transition in FSA_α has transitions for both $norm_i$ and $norm_{i+128}$. In addition to $FSA_\alpha, 1 \leq \alpha \leq r$, we have a section of FSA, FSA_{DT}, of length 128 corresponding to the decryption key.

Also, for every clause, we have a attack code substring of length 128 consisting only of byte 00h (the `NUL ASCII` character). Thus, the encrypted attack code for this given clause is $00 \cdots 0 \ \oplus \ k_0 \cdots k_{127} = k_0 \cdots k_{127}$.

The encrypted str_α is accepted by the FSA_α if and only if partial keys are chosen from one of the entries in the given key table in Table 9. Since every entry in the key table corresponds to an entry in the truth table of $clause_\alpha$, the encrypted str_α is accepted by the FSA_α if and only if $clause_\alpha$ is *true*.

The final FSA, FSA_{SAT}, and attack code corresponding to the 3-SAT problem are shown in Figure 30. It takes polynomial time to construct FSA_{SAT}. Please note that there exists a solution to the given PBA_{xor}^{FSA} problem if and only if the encrypted str_α is accepted by FSA_α for all $1 \leq \alpha \leq r$.

If the above PBA_{xor}^{FSA} problem has a solution encryption key key, then we can find assignments for variables $x_i, 0 \leq i \leq 127$ using Equation 29. Since S_{key_ac} is accepted

by FSA_{SAT} for key key, the encrypted str_α is accepted by FSA_α for all $1 \leq \alpha \leq r$. The encrypted str_α is accepted by FSA_α only if $clause_\alpha$ is *true*. Thus, all clauses are *true* and we have a solution for the 3-SAT problem.

Also, if there exists a 3-SAT solution assignment of variables x_i, then using Equation 27 we can compute the encryption key. All $clause_\alpha$, $1 \leq \alpha \leq r$, of the 3-SAT is *true* for the given variable assignment. Since $clause_\alpha$ is *true* only if the encrypted str_α is accepted by FSA_α, all encrypted str_α, $1 \leq \alpha \leq r$, are accepted by FSA_α. Thus, S_{key_ac} is accepted by FSA_{SAT}.

Thus, PBA^{FSA}_{xor} is as hard as 3-SAT. Since PBA^{FSA}_{xor} is also in NP, PBA^{FSA}_{xor} is an NP-complete problem. ∎

4.4.4.3 Corollaries

We have now proved that finding an encryption key that ensures S_{key_ac} is accepted by the FSA of an IDS is NP-complete. Suppose we allow the solution to have $\epsilon \| S_{key\_ac} \|$ number of invalid transitions, the problem still remains NP-hard because of the fact that $(1-\epsilon)$-SAT (or ϵ-UNSAT, $\epsilon < 1$) is an NP-hard problem.

Now consider the problem of finding an encryption key that optimally matches the S_{key_ac} to sFSA. This problem is considered harder than $PBA^{FSA}_{sub/xor}$ because in addition to the requirement of using only valid transitions of the sFSA, we need to match the probability of each transition in the sFSA. Thus, finding the suitable encryption key for sFSA should be NP-hard. Following similar logic above, we can also conclude that finding an encryption key which matches the S_{key_ac} to sFSA within ϵ bound of the optimal solution is also NP-hard.

Substitution and *XOR* are among the simplest encryption schemes. In fact, the more complex encryption schemes such AES and DES [25] use substitution and *XOR* as basic operations. Since we have shown that the problem is hard when the simpler schemes are in use, we can conclude that the problem is still hard when using the

other more complex encryption schemes.

To conclude, we have shown that finding an encryption key that ensures S_{key_ac} is accepted by the (s)FSA of an IDS is a hard problem (NP-complete).

4.4.4.4 Approximate Solutions

The most direct and important corollary is that the problem of generating a polymorphic blending attack by encrypting attack code is hard. In Section 4.4.4.1 and 4.4.4.2, we showed that the problem of finding an appropriate encryption key for a polymorphic blending attack is very hard. That is, it may take time exponential to the key length or character size. Although an attacker may not have any time restrictions, a polynomial time solution is clearly more desirable. There are good heuristic solvers available for the SAT problems or Integer Linear Programming (ILP) problems. These solvers provide approximate solutions in very reasonable amount of time. If we have a non-stochastic, or FSA based, IDS, we can reduce the polymorphic blending attack problem to a SAT problem. For a sFSA based IDS, we can map the problem to ILP.

Before we show the SAT or ILP reduction, we would reduce the problem of finding an appropriate encryption key to the problem of finding a path from the start state to accept states in a Directed Acyclic Graph (DAG). An edge in the DAG may have constraints of the form $k_i = j,\ j \in U$, where U is the set of all characters. The chosen path should contain minimal number of conflicting constraints. In addition, we may have some restrictions on the frequency of occurrences of edges corresponding to the frequency matching requirement for sFSA. In the following sections, we use the example shown in Figures 23 and 24 to illustrate the concept.

4.4.4.4.1 Construction of DAG Given a (s)FSA, a key length (l_k), and an attack code ac of length l_{ac}, we construct a DAG of depth $l_k + l_{ac}$ as follows. Suppose s_0 is the end state of the path in (s)FSA as traced by the attack vector and the decryptor. v_0 is the root vertex of the DAG corresponding to state s_0. At every

depth d of the DAG, we have a set of vertices $V_d = \{v_{d_i}\}$ such that the state v_i is reachable from state s_0 in exactly d transitions in (s)FSA. The accept vertices of the DAG are the leaf vertices at depth $l_k + l_{ac}$, which correspond to accept states in the (s)FSA. There exists an edge $e_{d_{ij}}$ from v_{d_i} to $v_{(d+1)_j}$ if and only if there exists a transition t_{ij} from state v_i to state v_j in (s)FSA. The weight of the edge is proportional to the probability of transition t_{ij}. The constraint $constr_{d_{ij}}$ associated with an edge $e_{d_{ij}}$ at depth d is shown below.

$$
\begin{aligned}
constr_{d_{ij}} &= (k_d == char_{ij}), && \text{if } d < l_k, \\
&= (k_{ac[d-l_k]} == char_{ij}), && \text{if } d \geq l_k \text{ and substitution}, \\
&= (k_{(d \bmod l_k)} \oplus ac[d - l_k] == char_{ij}), && \text{if } d \geq l_k \text{ and XOR}
\end{aligned}
$$

where $char_{ij}$ is the normal character corresponding to transition t_{ij} and $ac[i]$ is the attack character at the i position of attack code. An example construction of DAG is shown in Figure 31. The DAG corresponds to the example FSA and example attack shown in Figure 23 and 24, respectively.

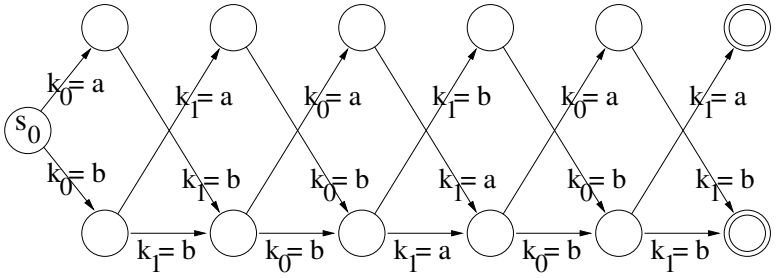


Figure 31: DAG corresponding to example FSA

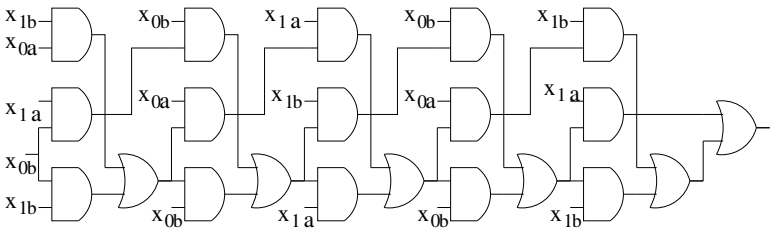


Figure 32: SAT representation of example DAG

The problem of finding an appropriate encryption key for a given attack code and FSA is equivalent to finding a path from the root vertex to an accept vertex in the DAG. Given a path P_{dag} in DAG consisting of edges with no (or minimal) conflicting constraints, we can find the encryption key by setting the constraints of the edges on the path to *true*. The path P_{fsa} followed by S_{key_ac} in the (s)FSA is similar to the path P_{dag}. If $e_{d_{ij}}$ is an edge at depth d in P_{dag} then transition t_{ij} is in P_{fsa} at depth d.

4.4.4.4.2 Translation to SAT If the given IDS is an FSA with no probabilities on transitions, the problem of finding a appropriate path in DAG can be translated to SAT. First, we translate the DAG problem to CIRCUIT-SAT [12]. Then we can efficiently translate CIRCUIT-SAT to SAT. For each constraint of the form $k_i = j$ in the DAG, we have a variable x_{ij} that is *true* if and only if the constraint is satisfied, and false otherwise. Now we can directly translate the DAG to CIRCUIT-SAT. A vertex v with input degree deg_{in} in DAG has a corresponding OR gate (OR_v) in CIRCUIT-SAT with deg_{in} inputs. For every outgoing edge (with some constraint $k_i = j$) of a vertex v, we have a AND gate whose input is x_{ij} and OR_v. The final output is OR of all the accept states. Figure 32 shows the conversion of our example DAG to CIRCUIT-SAT. We can then efficiently translate the CIRCUIT-SAT into a SAT problem. In the given SAT problem, we need to add additional requirement that a given key k_i is assigned to only one normal character. This means if x_{ij} is $true$ then $x_{ij'}$ is $false$ for all $j' \neq j$. Also, for substitution based encryption scheme, we need to add additional clauses in the SAT to ensure that a normal character is assigned to a single attack character. These cardinality constraints can be efficiently represented in SAT [53]. Furthermore, for a substitution scheme, there exists empty entries in the decryption key table corresponding to the characters not present in the attack code. These characters can be mapped to any unassigned normal character.

This requirement can be written as clause $\bigwedge_{j \in N}((\overline{\bigvee_{i \in M} x_{ij}}) \vee (\overline{\bigvee_{i \in \overline{M}} x_{ij}}))$, where M and N are the set of attack and normal characters, respectively. We can solve the final SAT problem using one of the several available SAT solvers, which are capable of solving huge SAT (or ϵ-UNSAT) problems in reasonable time.

4.4.4.4.3 Translation to ILP For a sFSA IDS, we propose to translate the problem of finding a good path in a DAG into a Integer Linear Programming problem. ILP is known to be NP-hard but there exist multiple good heuristics to solve big ILP problems. An ILP tries to find the minimum of a linear function over a set of variables defined by a finite number of linear constraints. All the variables in the solution should be integer. An ILP is called 0-1 ILP if all the variables are required to be either 0 or 1. Now we will show the reduction of finding the optimal path in a DAG to ILP problem.

For every edge $e_{d_{ij}}$ at level d in the DAG, we have a variable $h_{e_{d_{ij}}}$.

$$h_{e_{d_{ij}}} = 1 \qquad \text{if edge } e_{d_{ij}} \text{ is in the } \textit{solution path} \text{ of DAG,} \tag{30}$$

$$= 0 \qquad \text{otherwise} \tag{31}$$

For every constraint $x_i = j$ in the DAG, we have a variable $constr_{ij}$.

$$constr_{ij} = 1 \qquad \text{if constraint } x_i = j \text{ is } \textit{true} \text{ in the solution of DAG,} \tag{32}$$

$$= 0 \qquad \text{otherwise} \tag{33}$$

There is a valid path from start vertex to an accept vertex if and only if the number of *selected* outgoing edges at start state is one, the number of *selected* incoming edges is equal to the number of *selected* outgoing edges for all the intermediate vertices, and the number of *selected* incoming edges is equal to one for one of the end vertices. An edge is *selected* if it is in the solution path. These three conditions can be represent in terms of linear equations as follows:

$$\sum_{e \in OUT(v_0)} h_e + err_{v_0} = 1 \quad (34)$$

$$\sum_{e \in IN(V_{accept})} h_e + err_{accept} = 1 \quad (35)$$

$$\sum_{e \in IN(v_{di})} h_e + err_{di} = \sum_{e \in OUT(v_{di})} h_e, \forall\ v_{di} \text{ at depth } d, \forall 1 \leq d \leq l_k + l_{ac} - 1 \quad (36)$$

where $IN(v)$ and $OUT(v)$ is the set of in and out edges of vertex v, respectively. v_{di} is the ith vertex at depth d. The err terms account for invalid paths in the solution and are 0 if the path conditions are satisfied. In case, the condition is not satisfied, err_{di} can be either 1 or -1 depending on the difference of the number of *selected* incoming and outgoing edges at given node.

At any depth $d, 0 \leq d \leq l_k + l_{ac} - 1$, the number of edges from vertices at depth d to vertices at depth $d+1$ should be one. That is,

$$\sum_{e \in OUT(V_d)} h_e + err_{V_d} = 1, \forall 0 \leq d \leq l_k + l_{ac} - 1 \quad (37)$$

where V_d is the set of vertices at depth d. Again, the err_{V_d} term account for the errors. err_{V_d} can take values 0 or 1 depending on the number of outgoing edges at a given depth.

If an edge in the DAG is chosen in the path, the corresponding constraint should be satisfied. Suppose $constr_e$ represents the constraint associated with edge e. Then the requirement can be satisfied using following equation: $constr_e \geq h_e, \forall$ edge $e \in$ DAG.

Further, we can ensure that a given key is assigned to only one character by using following equation:

$$\sum_{j \in U} constr_{ij} = 1, \forall 0 \leq i \leq l_k \quad (38)$$

where U is the set of all possible characters and l_k is the key length.

For one-to-one byte substitution scheme, a normal character should not be assigned to multiple attack characters. That is,

$$\sum_{i \in M} constr_{ij} \leq 1, \forall j \in N \tag{39}$$

where M and N are the set of attack and normal characters, respectively. The following set of equations ensure that the characters not present in the attack character set are mapped only to normal characters not assigned to attack characters.

$$NAC_j \times \|\overline{M}\| \geq \sum_{i \in \overline{M}} x_{ij}, \forall j \in N, \tag{40}$$

$$NAC_j + \sum_{i \in M} x_{ij} \leq 1, \forall j \in N \tag{41}$$

The first equation makes sure that NAC_j is 1 if any non-attack character is mapped to a normal character j. The second equation ensures that if NAC_j is 1, then normal character j is not assigned to any attack character, and, vice-versa.

The above set of equations guarantee that there exists a path from the start vertex and the end vertex with some errors. Now we present the minimization criteria to reduce the errors and the distance of S_{key_ac} from the sFSA. Assume a distance metric of the form:

$$dist = \sum_{transition\ t \in sFSA} \kappa_t \times |p_t - \frac{num_t}{l_k + l_{ac}}| \tag{42}$$

where κ_t is some constant associated with transition t, p_t is the probability of the transition to be taken in sFSA, and num_t is the number of times the transition t is taken by the S_{key_ac}.

The minimization criteria for the ILP problem can be then written as:

$$\sum_{transition\ t\ \in sFSA} const_t \times |p_t - \frac{\sum_d h_{d_t}}{l_k + l_{ac}}| + \sigma \times (\sum_{v \in V_{dag}} |err_v| + \sum_d err_{V_d}) \tag{43}$$

where V_{dag} is the set of vertices in the DAG and σ is the weight of the errors caused by taking a invalid transition in the sFSA. Note that the $|\alpha - \beta|$ term in minimization can be rewritten as abs_{diff} where, $\alpha - \beta \geq -abs_{diff}$ and $\alpha - \beta \leq abs_{diff}$.

Solving the above ILP for the given minimization criteria will provide the encryption key. Using this we can generate the encrypted attack code and prepare the polymorphic blending attack packet by concatenating attack vector, decryptor, decryption key, and the encrypted attack code. We can then perform padding to match the final attack packet even closer to the normal.

4.4.4.4.4 Heuristic Solutions Rather than finding the optimal solution, an attacker may simply apply some heuristics that produce an approximate (or good enough) solution using much less resources (time and memory). Here we present a simple heuristic that finds a good approximate solution very efficiently.

The heuristic is based on the *hill climbing* search algorithm, which is used widely in artificial intelligence and constraint solving. *Hill climbing* starts with an initial solution and iteratively improves it. At each step, the algorithm looks at neighboring solutions and choose one that is better than current solution. The definition of neighbors depends on the problem domain.

We now present our heuristic. Given an IDS and an attack instance, we choose a random encryption key and calculate the distance between S_{key_ac} and (s)FSA. Now, we randomly choose a k_i to modify in the key. For all the possible character (c) values, we first temporarily assign $k_i = c$. For a substitution scheme, if c is already assigned to some attack character k_j, we temporarily swap the normal characters assigned to k_i and k_j. We then find the new distance to (s)FSA using the temporary key. We choose the character that reduces the distance by the maximum value and assign it to k_i. At this point, a new key position (k_j) is chosen to modify and the process is repeated. This above process is iterated for the desired number of iterations or till a satisfactory solution is produced.

It is possible in the above approach to reach a local maximum that is not very close to the optimal solution. We reach a local maximum if modifying any key increases the

distance to (s)FSA. To overcome this problem, whenever we reach local maximum, we choose a small set of key positions and set them to some random values, and restart the above iterative process of finding solution. The idea is by randomly picking another starting point in the solution space, the new solution point may belong to a locale that has better local maximum.

The above heuristic can give us very good solution depending on the number of iterations. The pseudo-code for the heuristic is shown in Appendix B.4

4.5 Experiments and Results

The motivation of our research was to address the open problem: given an anomaly detection system and an attack, can one *automatically* generate the PBA instances? Thus, in our experiments, we wanted to directly compare our formal framework with the more ad-hoc approaches developed before. The key elements of our experiment set-up were the same as in Section 4.2. We used the same anomaly detection systems, namely, PAYL 1-gram and 2-gram, as well as the same attack and same traffic datasets, as in Section 4.2.6.1. The results showed that, although our framework is based on an abstract model of IDS and uses general algorithms, it automatically generated PBA instances that were more evasive (i.e. better matched the IDS normal profiles) than or at least as good as the PBA instances from the more PAYL specific algorithms in Section 4.2.

Briefly, the experimental dataset contains 7 days of Web traffic with 4.7 million packets. 4.3 million packets were used for training the IDS profile. A part of the remaining traffic was used to generate/train an artificial profile used by the attacker. We generated PAYL 1-gram and 2-gram models (using the 14 days of traffic) for three different packet lengths, namely, 418, 730, and 1460. The attack vector is based on the implementation of `firew0rker` [18]. More information of the dataset can be found in Section 4.2.6.

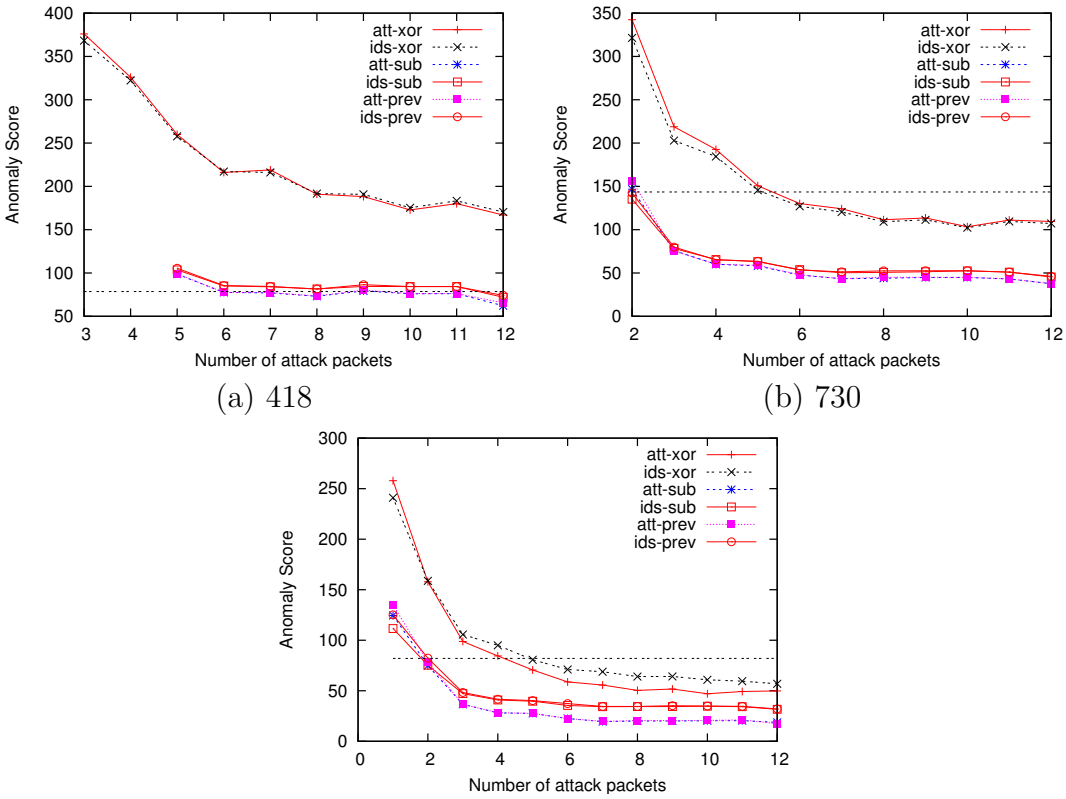
For all the experiments, we divided the attack flow into multiple packets. The attack vector was placed at the start of the first packet. The decryptor was divided into several sections and allotted to different attack packets. The attack body was also divided into multiple chunks. The sFSA corresponding to the artificial profile was adjusted for attack vector and polymorphic decryptor. A separate encryption key for each attack body fragment was generated using our framework to match the adjusted artificial profile. Each attack body fraction was encrypted using the corresponding key and appended to the corresponding attack packet. Then each attack packet was padded to the desired packet length. The final attack packets were then used together to launch an attack.

4.5.1 PAYL 1-gram Evasion

We applied our framework to generate polymorphic blending attacks to evade 1-gram PAYL, using substitution-based encryption and XOR encryption, respectively. For the XOR scheme, we used a 64 byte key. For each encryption scheme, we translated the problem of finding the optimal encryption key for 1-gram evasion to an ILP problem. We used *ILOG CPLEX* to solve the ILP problems. *CPLEX* is a commercial optimization tool for solving Mixed Integer Programs (MIP). We obtained a near-optimal solution (i.e., encryption key) for the ILP problems. The attack code was then encrypted using this key, and padding was performed. For comparison, we also generated polymorphic blending attacks using the greedy one-to-one local substitution scheme presented in Section 4.2.3.2.1.

It took 6.5 seconds on average to solve an ILP problem on a Pentium-M 2GHz machine. The solution provided was within 0.2% of the optimal solution. Figure 33 shows the distance of the attack flow from the artificial profile and the IDS profile. The results for both substitution and XOR encryption schemes, as well as the greedy scheme from Section 4.2.3.2.1, are shown in the figure. The x-axis shows the number

of packets attack flow was divided into and the y-axis shows distance of the attack flow from the artificial profile and IDS normal profile. This distance is the maximum of the distances of individual attack packets in a flow. A horizontal line corresponding to anomaly error threshold for the 1% IDS false positive is also shown.



(a) 418 (b) 730

(a) 1460

Figure 33: Anomaly score or error distance of 1-gram blending attack. The plots with prefix *att* and *ids* corresponds to distance from the artificial profile and the IDS profile, respectively. *xor* and *sub* corresponds to the PBA generated for XOR and substitution based schemes using our framework. *prev* denotes the algorithm from previous paper.

The error distance of attack generated for substitution based encryption using ILP is almost identical to the greedy approach. Thus, the greedy 1-gram blending approach also provides a near optimal substitution table.

The error distance for attacks generated using the XOR encryption scheme is much higher. This is expected. For substitution, by replacing attack characters with normal characters, we can ensure that only normal characters are present in

the mutated attack packet. For *XOR*, it is harder to find an appropriate key such that it contains only normal characters and *XOR*ing it with attack characters also results in only normal characters. For packet length 418 and 730, the error distance of PBA generated using XOR based scheme is twice or more than the substitution-based scheme. For packet length 1460, the error distance for XOR based scheme is comparatively smaller. Also, the difference decreases as the number of attack packets in the attack flow increases. The large amount of padding space available masks the error produced by the attack code in XOR based scheme.

In the graphs, all the attack points below the horizontal error threshold line will not be detected by the IDS with a 1% false positive rate. If the false positive rate is decreased, typically the anomaly error threshold is increased. That is, the horizontal line is moved up, and more attack points will be missed by the IDS.

For packet length 730 and 1460, we need only two packets to evade PAYL 1-gram when using substitution. The IDS can be evaded using attack flow of size as low as 1460. For packet length 418, we need 8 packets to evade the IDS. The XOR based scheme can also evade the IDS for packet lengths 730 and 1460, although with bigger attack size. For packet length 418, XOR needs to divide attack packets into many more number of packets in order to evade the IDS.

4.5.2 PAYL 2-gram Evasion

We also generated polymorphic blending attacks to evade PAYL 2-gram. We used the heuristic presented in Section 4.4.4.4.4 to generate such attacks. We started with a random solution and iterated the hill climbing steps 25000 times. The best encryption key seen during the process was recorded. The attack code was then encrypted using this key and the attack packet was padded to the desired length. The distance of the attack flow from the normal profiles was recorded. For comparison, we also generated the polymorphic blending attacks using the greedy 2-gram blending

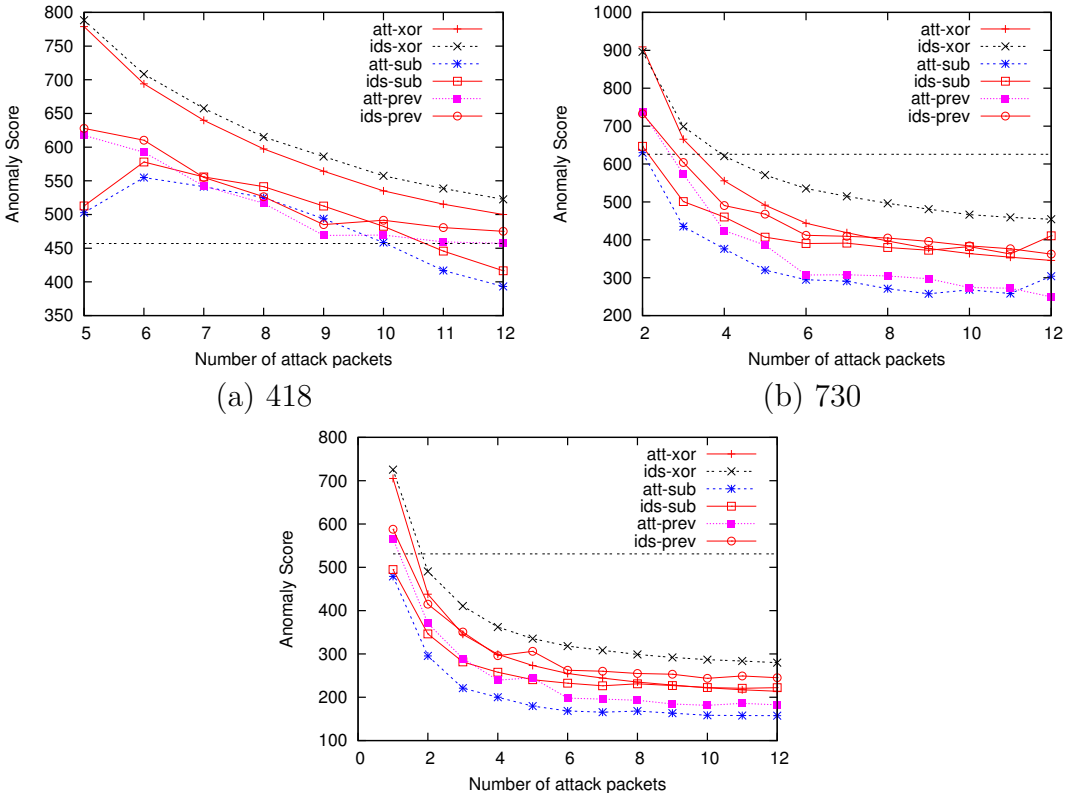
algorithm presented in Section 4.2.4.2.

For substitution based encryption, it took $10min$ on average to perform 25000 iterations on a given problem. For XOR encryption, performing 25000 iterations took little more than an hour on average. The time of each iteration is dependent on the range of keys and the number of terms in the distance calculation. For substitution, the range of keys is all the normal characters; whereas in XOR, the range of keys can be set of all the possible characters. Also, since XOR is not able to match the normal profiles closely, there several new 2-grams in the attack packet and thus the number of terms in the distance calculation may become big. These two reasons attribute to long running time for XOR.

The distances of the attack packets from the normal profiles are shown in Figure 34. The results for substitution-based encryption and XOR encryption, as well as the greedy scheme are shown, along with a horizontal line corresponding to error threshold of the 1% IDS false positive rate.

Using substitution, our heuristic-based approach is able to better match the normal profile than the previous greedy approach for most attack instances. For packet length 418, when the number of attack packets is 8 or 9, the previous approach matches the normal profile better than the heuristic from our framework. Checking the distance of individual attack packets, we observed that for some packets, the heuristic got stuck in a bad local maximum for considerable number of iterations. Thus, the heuristic was not able to find a good solution in the given number of iterations. In such cases, one can restart the heuristic using another random solution or run the heuristic for more number of iterations.

Compared with PAYL 1-gram, we needed more number of attack packets to evade PAYL 2-gram. The minimum attack size required to evade the PAYL 2-gram is 2190, or 3 packets of length 730.



(a) 418 (b) 730

(a) 1460

Figure 34: Anomaly scores of 2-gram blending attacks.

4.6 Countermeasures

The experimental results reported above show that the statistical models used by current network anomaly detection systems are not sufficiently accurate in detecting deliberate evasion attempts. By following the ideas presented in this thesis, it may be fairly easy to devise different blending algorithms in order to evade other network anomaly IDSs that rely solely on packet statistics.

Given an IDS that is vulnerable to a polymorphic blending attack, we want to identify the shortcomings of the IDS and improve the IDS so that these evasive attacks are harder to generate. It is desirable to modify the IDS so that the IDS is resistant to blending attacks generated using any attack vector, attack code, or decryptor. In the following section, we present the drawbacks of current anomaly IDSs and steps that can be taken to improve the robustness of an IDS.

4.6.1 Drawbacks of Current Anomaly IDSs

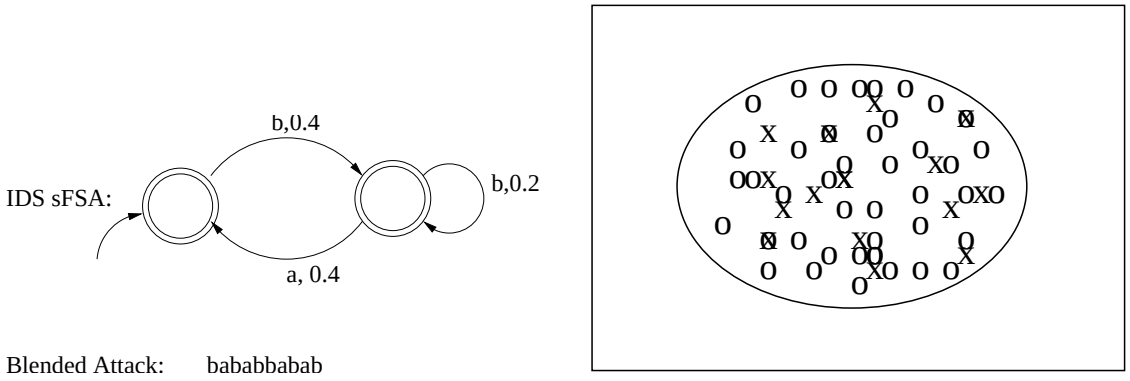
As discussed earlier, current anomaly IDSs can be modeled as an (s)FSA. An IDS also uses a classifier to discriminate attacks from normal data. The input features used by the classifier are the transition frequencies of each transition in the (s)FSA. The existence of evasive attacks can be attributed to shortcomings in the FSA model or features. These attacks may also be possible due to unsuitable classifiers or distance-metrics used by the IDS. Some of the drawbacks of anomaly IDSs are:

- Imprecise (s)FSA model
- Non-discriminating and noisy features
- Distance-based classifier
- Deterministic algorithm
- Single feature set

4.6.1.1 Imprecise (s)FSA model

Anomaly detection systems presented in Section 4.3.1 use simple statistics or packet structure to represent the normal traffic. Since any service provided over a network follows certain application level protocols, a normal request to the service should follow the given protocol syntax and semantics. These application syntax and semantics related information cannot be modeled accurately using simple statistics of network packets. Therefore, the simple statistics, and hence the (s)FSA, used by a payload network anomaly IDS is an imprecise representation of normal traffic. It may be possible to match the statistics using a polymorphic blending attack instance. Figure 35 shows a hypothetical scenario where the (s)FSA model is not precise enough to detect polymorphic blending attacks. Attacks match the normal profile very closely and overlap with normal data in the feature space. In the given scenario, the (s)FSA

model used by the IDS is not sufficient to detect PBAs. We need a new (s)FSA model to detect PBAs.



(a) FSA model view (b) Feature space view

Figure 35: Example of an imprecise sFSA model. The circles represent normal data points and the crosses represent attack data points. The rectangular box is the complete feature space. The ellipse represents the normal boundary.

Instead of using simple statistics, an IDS should use normal protocol semantics. By using more protocol semantics and syntactic information, the IDS can represent the structure of normal traffic more precisely. It should be harder for an adversary to compromise a system while following the normal protocol semantics. Some protocol-based, or equivalently specification-based anomaly detection systems proposed in the IDS research community are [65, 52]. Protocol-based anomaly detection systems ensure that the monitored traffic follows certain protocol semantics. They also check if some statistical patterns of these semantic information match the normal statistics. The protocol-based anomaly detection systems are effective in detection of attacks which misuse protocols.

However, modeling protocol syntax and semantic is computationally more expensive than measuring simple traffic statistics. A typical protocol-based IDS requires parsing the traffic, checking the syntax of the packets, maintaining a state to follow the status of the flow, and matching the semantics. This process requires a lot of memory and processing time. In addition, a protocol-based anomaly IDS needs to know all the applications running in the network along with their detailed protocol

specification. This may be infeasible if the system provides multiple services, or if the protocol semantics are frequently modified. Furthermore, the protocol semantic patterns learnt from a given normal dataset do not generalize well for previously unseen normal access to the service. Thus, such an IDS may produce a lot of false positives. Therefore, the trade-off between detection accuracy, hardness of evasion, operational speed, and the IDS management has to be considered.

4.6.1.2 Non-discriminating and noisy features

Not all the features in a given feature set are useful for detection purposes. A feature may be non-discriminating; that is, the value of the feature is the same for both the normal data and attack data. It may also be noisy with very high standard deviation. These features can actually reduce the IDS accuracy. For example, in case of n-gram PAYL, there can be some n-grams whose frequencies are inconsistent throughout different normal packets and may force the IDS to have a higher error threshold. Thus, a PBA that was above the error threshold and detectable by the IDS otherwise, may not be detectable if we use these noisy n-grams. An IDS should perform feature extraction and select a set of features that are useful in differentiating attacks from the normal.

Principal Component Analysis (PCA) is commonly used for feature extraction. PCA is useful only if data from both classes are present. Initially, the features are de-correlated using *Karhunen-Loeve Transform (KLT)*. PCA assumes that the difference between classes are contained in the high variance features. To improve the performance of the classifier, only the features with high variance are chosen and low variance features are removed.

The anomaly IDS is a one-class classifier and data from only normal class is assumed to be available. Unlike a multi-class classifier, the feature extraction principle used for PCA does not hold true and high variance features may not necessarily be

good in finding outliers. Tax and Muller [60] showed that for one-class classification, features with low variance features are informative and are better for finding outliers. To improve the IDS, we should choose low variance features. The less noisy features set should be better in detecting polymorphic blending attacks.

4.6.1.3 Distance-based classifier

Once an IDS has determined the value of features (or state transition frequencies), it uses a classifier to calculate its distance from the original set of normal features. Most of the current proposed IDSs use a very simple distance-based classifier similar to the *Minkowski distance metric*. These classifiers use a weighted sum of deviation of features from the mean normal value of the feature. They also assume that the features are independent. This assumption significantly reduces the complexity and the overhead of the distance calculation. But due to their simple design and flawed assumptions, these classifiers may not be well suited for anomaly detection.

Furthermore, the classifier performance depends on the characteristics of the data. The distance-based classifiers used by anomaly IDSs work well if the normal data follows *Gaussian* distribution and the attack data is linearly separable from the normal data. It is possible for the normal data to follow different distributions which may not be suitable for such distance-based classifiers. Thus, these classifiers may not differentiate between the normal data and attacks satisfactorily.

Therefore, instead of using a simple distance-based classifier, an anomaly IDS needs to use more powerful classifiers. Some of the powerful and complex classifiers are decision trees, neural networks, bayesian networks, hidden markov models and support vector machines. There is no single classifier that works best for all given problems [49]. Thus, an IDS designer needs to determine the data characteristics and use empirical tests to find the best classifier. Some of these classifiers are high resource consuming and may not be suitable for real-time detection purposes. Again

we need to find a middle-ground between run time resource overheads and detection accuracy.

4.6.1.4 Deterministic algorithm

Most of the current proposed anomaly IDSs use a deterministic algorithm or classifier to detect attacks. We assume that the adversary knows the complete detection mechanism, including the learning algorithm, and the detection algorithm. In polymorphic blending attacks, the adversary uses these information along with the blending algorithms presented here to generate attacks that are able to evade the IDS.

A possible countermeasure is to introduce randomness in the IDS model. Consider an IDS that uses multiple profiles, say X number of profiles, to represent the normal traffic. These normal profiles are generated using different classifiers and different sets of features, that are at least partially independent. Suppose the IDS also uses an input parameter that is generated randomly. Based on the value of this random parameter, the IDS may use different feature sets and classifiers to detect intrusions. Since the adversary does not know the value of these parameters, the adversary has two options while generating a polymorphic blending attack.

1. First, the adversary may try to guess which profile is being used currently by the IDS for detection and may try to generate a polymorphic blending attack to evade that particular normal profile. If the adversary guesses correctly, he/she has a high chance of evading the IDS. But if the adversary guesses wrong, then given that the features used in different normal profiles are different and partially independent, there is a small chance that the attack might match the normal profile currently used by the IDS. Thus, with high probability, the attack will be detected by the IDS. Since the probability of the adversary guessing correctly is small (around $\frac{1}{X}$), the overall chances of the adversary succeeding in evading the IDS is small.

2. The adversary may try to match all the normal profiles in order to ensure that the attack is not detected, no matter which normal profile is being used by the IDS. But matching all the profiles may be significantly harder than matching a single profile.

Thus, in both cases using randomization makes it harder for the adversary to evade the IDS. Unlike other countermeasures, using randomization does not have any evident performance dis-advantages. But determining sets of features that are partially independent and are also sufficient in detecting attacks may not be easy. One needs to take care in ensuring that each randomized model is adequate in detection of traditional attacks.

4.6.1.5 Single feature set

Anomaly IDSs discussed in Section 4.3.1 use a single feature set to represent a given packet or a section of a packet. For example, PAYL uses n-gram distribution of the whole payload to represent normal payloads. [31] uses a statistical feature or structure to represent a given section of normal traffic. Therefore, for each section, the adversary is required to match a single statistical model to evade the IDS.

A better defense approach is to use multiple IDS models that use independent features to record normal profiles. A monitored packet is normal if and only if the packet matches all or a majority of the models. Therefore, in order to evade detection, a polymorphic blending attack created by the adversary will need to match all (or the majority) of the models. This can be significantly harder than matching a single model if features used for different models are independent. Independent features ensure that matching one model does not automatically imply matching other models. An IDS using multiple models [43] is shown to be effective in detection of polymorphic blending attacks.

Using multiple models for detection requires significantly more computing resources, including cpu and memory, as compared to a single model. Also, multiple models may be restrictive and a new unseen normal pattern may not be accepted by all or a majority of the models. Thus, we expect the IDS to have a higher false positive rate.

4.6.2 Improving the Robustness of an (s)FSA IDS

To improve the robustness of an (s)FSA IDS, we need to identify the shortcomings of the IDS and improve on it using the principles presented in Section 4.6.1.

4.6.2.1 Improving (s)FSA

Figure 35 shows an example vulnerable sFSA IDS and a blending attack for the IDS. Since the PBA matches very closely to the sFSA, irrespective of the classifier used, the IDS will not be able to detect the attack. Therefore, the sFSA used by the IDS is not sufficient to detect the attacks. The IDS needs to improve the sFSA to detect the attacks.

A possible approach to improve an (s)FSA model is to modify the current (s)FSA by deleting transitions from the (s)FSA. First, we collect many attack instances for different known attacks on the system. For each of these instances, we generate several polymorphic blending attacks using techniques described in previous sections. We find transitions in the sFSA that are traversed frequently by many polymorphic blending attacks. Finally, we delete these transitions from the (s)FSA. The new modified (s)FSA should be able to detect the PBAs that were generated earlier. However, there may be lot of normal packets taking these removed transitions. Therefore, the modified (s)FSA IDS may have high false positive rate. Furthermore, once the adversary knows which transitions have been removed from the (s)FSA, she may generate PBAs that do not traverse the deleted transitions and thus the modified (s)FSA may not detect new PBAs. Therefore, modifying an (s)FSA is not a judicious

choice to improve the robustness of the IDS. An IDS developer needs to completely redesign the IDS by choosing better protocol semantic features.

4.6.2.2 Feature Extraction

Once we decide to use a given (s)FSA model to detect attacks, we can improve the features used by the IDS classifier. We can remove the features that are noisy or not useful in discriminating attacks from the normal.

If we do not have attack data available, we can perform feature extraction by performing *Karhunen-Loeve* transform on the normal training data and selecting low variance features. Given the initial set of features, we calculate the correlation matrix. Eigenvectors and corresponding eigenvalues are calculated for the given correlation matrix. The low variance features are the eigenvectors with smaller eigenvalues. Therefore, we choose the eigenvectors with small eigenvalues. We can use this approach to improve the IDS shown in figure 36. Instead of using two features f_1 and f_2, we can perform *Karhunen-Loeve* transform for the normal dataset, and get two new uncorrelated features, f_1^{new} and f_2^{new}. Since f_1^{new} is the low variance feature, we use it for detection. For the same false positive rate, the new IDS using single feature is better able to detect PBAs as compared to the original IDS.

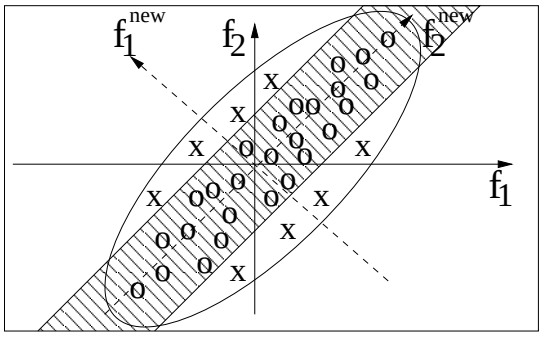


Figure 36: Example of feature extraction. Unlike the original normal boundary (ellipse), the normal boundary for the new IDS (shaded area) does not contain any attacks.

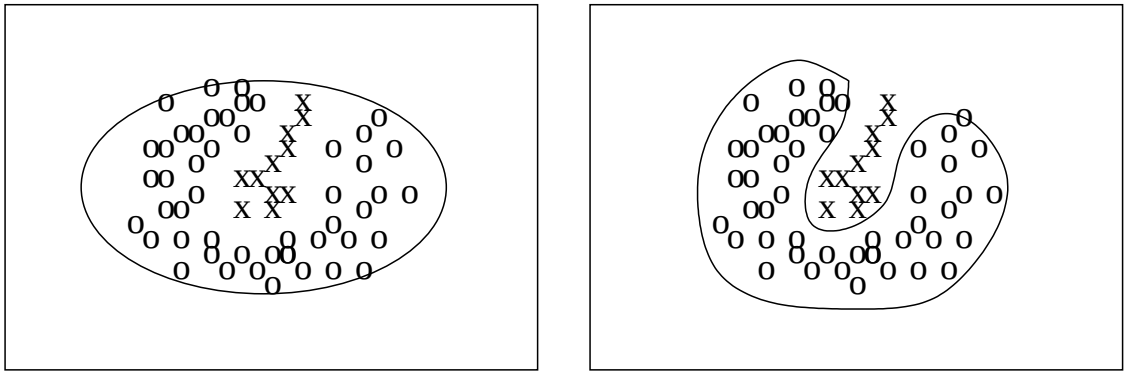
In case the attack data set is available, it is possible to use PCA and select discriminating features. First, we collect many attack instances for different known attacks on the system. For each of these attack instances, we generate several polymorphic blending attacks, using techniques described in section 4.4. We use these PBAs along with the normal training data and find new set of features using *Karhunen-Loeve* transform. Then, we choose features that have high variance. Selected features should be effective in detecting PBAs generated using the attack vector and the attack code used during PCA. However, this approach may not be helpful in detecting PBAs generated using other attack vectors and attack codes. In fact, choosing high variance features may actually reduce the detection capability of the IDS.

4.6.2.3 Improved classifier

Suppose features used by an IDS are not noisy and are good for discriminating attacks from the normal. But due to the classifier's limitation, the IDS may not be able to discriminate attack from the normal. An example of such a classifier problem is shown in figure 37(a). The data points and the PBAs are mapped to mutually separate spaces. However, they are not separable by an IDS which uses a simple distance-based classifier. In this scenario, to improve the robustness of the IDS, we need to use a powerful classifier. Since it is not possible to use a classifier that works best for all types of features and datasets, we need to experiment with several classifiers and choose the one that works best for the given training dataset and training features. The robustness of the IDS shown in figure 37(a) can be improved by using a better classifier that uses a different decision surface as shown in figure 37(b).

4.6.2.4 Randomization

There are several ways to include randomization in the IDS to make it harder for an adversary to generate PBAs. Some of the randomization techniques are:



(a) Original Classifier (b) Improved Classifier

Figure 37: Example of classifier improvement.

- *Feature Selection:* From a given feature set, we can randomly choose a subset of features to be used by the classifier for detection. These subset of features should be adequate to discern attacks from the normal. Features can be chosen from the original feature set or from the transformed (for example Karhunen-Loeve transform) feature set.

- *Merge features:* We can randomly merge features from the original feature set to obtain a new feature set. The new features can be calculated as linear combination of chosen features. Using a pseudo-random number generator, we can choose a certain number of features from the original feature set and merge them. We can also use a hash function to hash original features and merge features which have the same hash-value.

  A preferred method to merge a set of features is to use Karhunen-Loeve transform on the original subset of features and choose the transformed feature with minimum variance.

- *Distance calculation:* We can use different randomization techniques for distance calculation. An IDS can randomly choose a classifier from a set or pre-chosen classifiers that are known to work well. Also, the IDS can use randomization by tuning the parameters used by the classifier. In these cases, randomization

is used to exploit the different generalization capabilities of different classifiers. This approach may not be effective as different models still use the same set of features.

- *Data Sampling:* Another dimension of randomization is to monitor different parts of the data. Unlike traditional IDSs which monitor the complete data payload, we can create multiple models, each representing the normal pattern of a particular portion of the payload [70]. During monitoring, we can use a subset of these models for detection. Since the adversary does not know which portion of the payload is monitored by the IDS, the adversary needs to ensure that each portion of the attack packet matches the normal profile.

4.6.2.5 Multiple feature sets

To improve a given (s)FSA IDS, we generate a multiple subset of features that are independent. We cannot obtain completely new sets of features that are independent of current feature set because that will require further knowledge of the normal traffic. Instead, we can divide the initial feature set into multiple subsets of features that are uncorrelated and possibly independent. First, we take the initial features and de-correlate them using *Karhunen-Loeve* transform [14]. We can also use *Independent Component Analysis* to get a new feature set that is independent. Then, we divide the transformed features into multiple independent or uncorrelated subsets. This approach is useful if each individual transformed feature subset is sufficient for detecting intrusions.

In figure 38, the IDS using both the features cannot detect PBAs for a given false positive rate. However, suppose the IDS creates two models using feature ${f_1}^{new}$ and ${f_2}^{new}$, respectively. Also, suppose a packet is considered normal if and only if the output of at least one of these two models is normal. Thus, using multiple models with different feature sets, the IDS will be able to detect the attacks while maintaining

same false positive rate. This improvement approach is useful in the cases where the adversary cannot match individual features very closely.

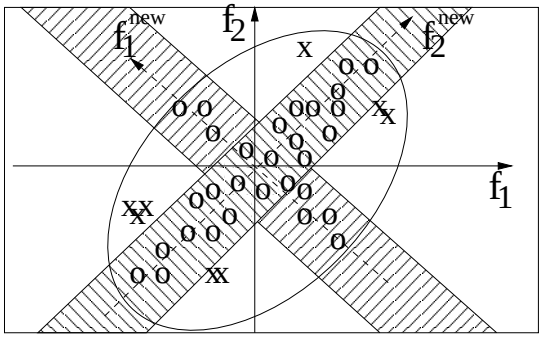


Figure 38: Example of IDS improvement using multi classifier. To remove the false positive, IDS considers area under ellipse as normal. However, using two model with features using ${f_1}^{new}$ and ${f_2}^{new}$, respectively, IDS can ensure that only shaded area is normal and PBAs lie outside the normal.

Please note that, depending on the characteristics of polymorphic blending attacks and quality of the feature set (or the (s)FSA model), it may or may not be possible to improve the robustness of the IDS using the techniques discussed above.

4.6.3 Experiments

We performed experiments to determine the effectiveness of different improvement techniques presented in Section 4.6.2. We used PAYL 1-gram and 2-gram anomaly detection systems for our experiments. We compared the robustness of improved PAYL IDS with the original PAYL IDS. We also measured the monitoring time of improved PAYL and compared it with the original PAYL to demonstrate the performance trade-off of different improvement techniques.

4.6.3.1 Experiment Setup

4.6.3.1.1 Dataset We used the same training dataset as in Section 4.6.2. The dataset contained 7 days of traffic coming to the `CoC` web server. Normal data packets from five hosts/LANs were chosen to train the artificial profile used by the adversary. Out of the rest of the data, 1 day of traffic was used to train the IDS normal profile

and the remaining 6 days of traffic were used to test the IDS. We used 7-fold cross-validation to verify the results of our experiments.

4.6.3.1.2 Blending Attack Three different attacks were used for the experiment. We used the windows media services attack (MS03-022) and same attack body as discussed in Section 4.2.6.1.1. In addition, we used two attacks on *Windows NT IIS* based on `DDK` and `CodeRed` attacks. `DDK` attack exploits the buffer overflow vulnerability present in Windows IIS (MS01-033). `CodeRed` also exploits a similar vulnerability (MS01-044) present in Windows IIS.

From our previous experiments, it is evident that the packets of length of 1460 are best suited for polymorphic blending attacks. Therefore, we divided the attack into multiple packets of length 1460. Attack vectors were placed at the location required by the given attack. The decryptor was divided into several sections and allotted to different attack packets. The sFSA corresponding to the artificial profile was adjusted for the attack vector and the polymorphic decryptor.

Also, since substitution based attacks are better in matching normal profile, we generated blending attacks using one-to-one byte substitution. For 1-gram PAYL, PBAs were generated using the greedy algorithms presented in Section 4.2.3.2.1. For 2-gram PAYL, heuristics discussed in Section 4.2.4.2 were used to generate PBAs. The attack body was divided into multiple fragments. A separate encryption key for each attack body fragment was generated to match the adjusted artificial profile. Each attack body fragment was encrypted using the corresponding key and appended to the corresponding attack packet.

Finally each attack packet was padded to the desired packet length. We used the greedy algorithms presented in Section 4.2.3.1 to generate a suitable padding. The final attack packets were then used to launch an attack.

The polymorphic blending attacks were generated for the original PAYL 1-gram

and 2-gram IDS. Since the number of training packets used to train the artificial profile is small compared to one used to train the IDS, the artificial profile is not accurate. Therefore, during the feature extraction, or support vector calculation in SVM-classifier, or other feature manipulation, the extracted features or the support vectors are significantly different than the ones generated for the IDS. Due to this reason, PBAs generated using the improved IDS algorithm matches worse to the IDS profile as compared to PBAs generated using the original PAYL. Therefore, we present the results for PBAs generated using the original PAYL IDS. Also, the results presented here are the average of results from 5 different hosts/LANs used for the training of the artificial profile.

4.6.3.2 Improvements on PAYL

We use techniques discussed in Section 4.6.2 to improve the robustness of PAYL 1-gram and 2-gram IDS. We used feature extraction to extract good features from the original feature set. We also improved on the simple distance metric used by PAYL. We also experimented with different randomization techniques to further improve on the original PAYL.

4.6.3.2.1 Feature Extraction To remove noisy n-gram features used by PAYL, we perform *Karhunen-Loeve* transform and chose the low variance direction. The initial set of features used for the transformation consists of all the frequently occurring n-grams along with one extra feature. The extra feature corresponds to all the n-grams that are rare or absent in the training dataset. We calculated the correlation matrix of the input features using the training dataset. Then we calculate the eigenvectors and the eigenvalues for the correlation matrix. Then we choose 0.2 fraction of eigenvectors with the smallest eigenvalues as the new set of features and used them for intrusion detection.

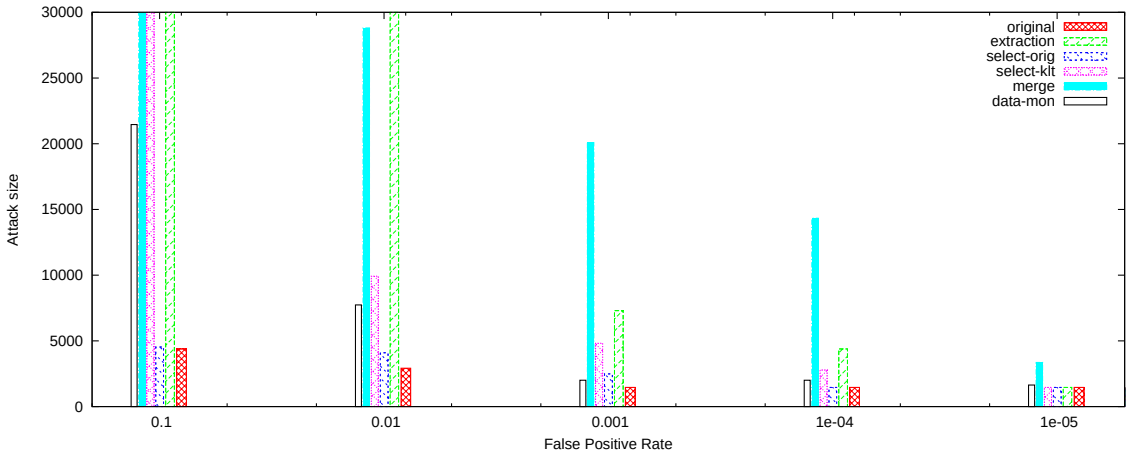
4.6.3.2.2 Randomization We used four different randomization techniques to make PAYL more robust. Each randomized method was repeated 10 times for validation purpose. The average of the 10 iterations is reported in the results.

- **Basic Feature Selection:** We randomly choose half of the original features (that is 50% of the seen n-grams) and used them for detection.
- **Transformed Feature Selection:** We also performed random feature selection after Karhunen-Loeve transform. After finding the eigenvectors and eigenvalues of the correlation matrix, we removed very high-variance eigenvectors. Of the remaining eigenvectors, we randomly choose half of the transformed features and used them for detection.
- **Feature Merging:** We randomly merged n-grams to obtain a new feature set. To merge a given subset of features into one feature, we performed Karhunen-Loeve transform and chose the eigenvector with the smallest eigenvalue. All the new features were generated by merging a fixed number of n-grams. The n-grams to be merged were chosen using pseudo-random number generator.
- **Randomized Data Sampling:** The payload was divided into multiple sections and for every instantiation of IDS, a random sections was monitored.

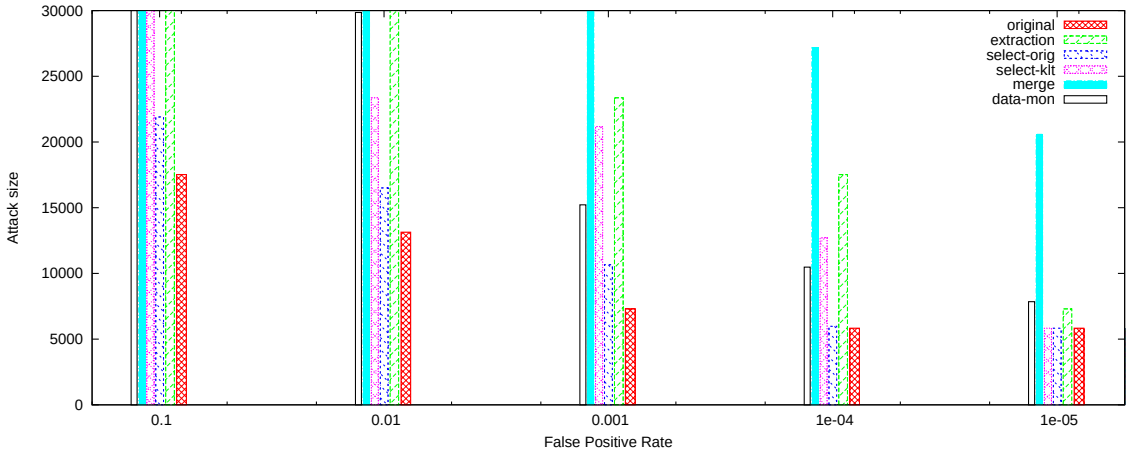
4.6.4 Results

Improved IDSs along with original n-gram models were first tested using traditional polymorphic attacks. All IDSs were able to detect all the traditional polymorphic attacks. Next we generate polymorphic blending attacks to test the robustness of our IDSs.

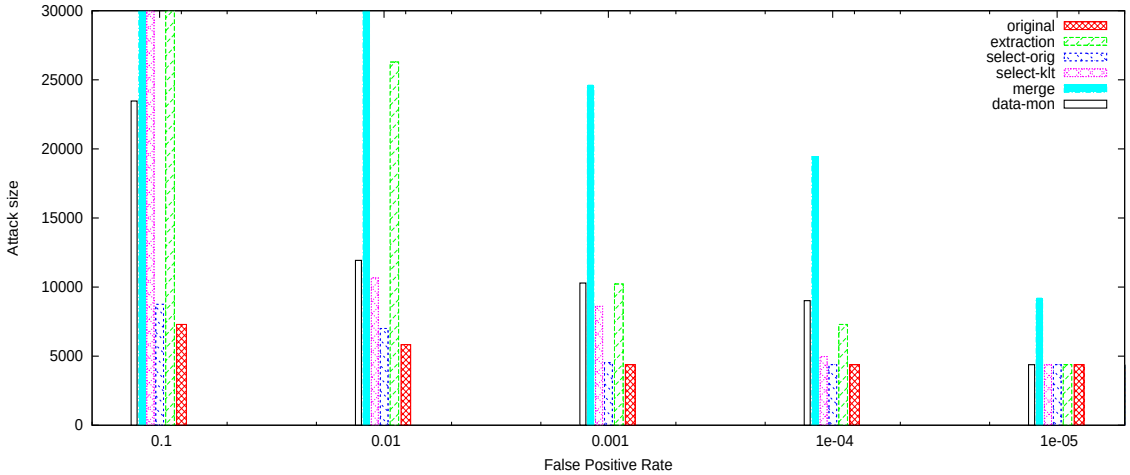
Figure 39 shows the size of the blending attack required to evade improved 1-gram PAYL anomaly IDSs proposed in Section 4.6.3.2. Bigger attack size implies more robust IDS. Bar touching the $30K$ attack size indicates that none of the blending



(a) Windows media services attack

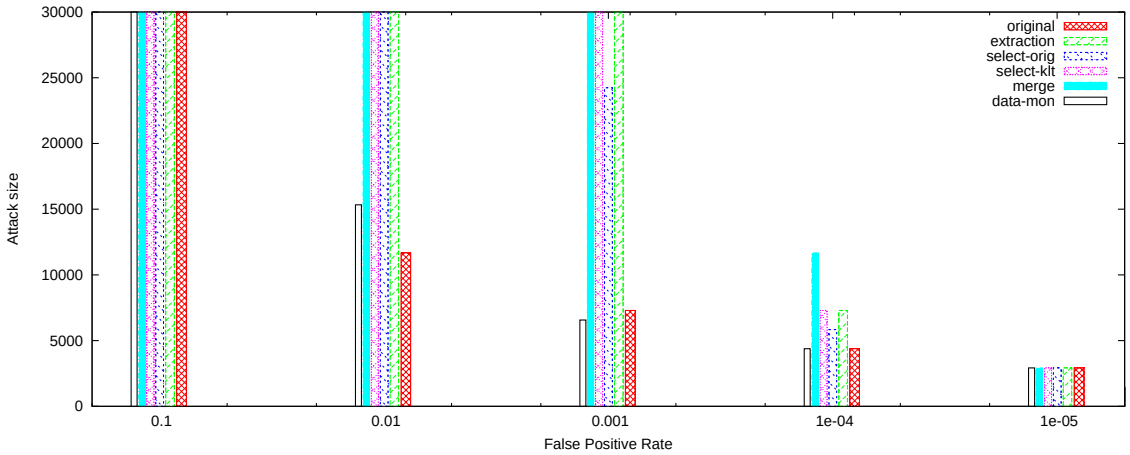


(b) CodeRed attack

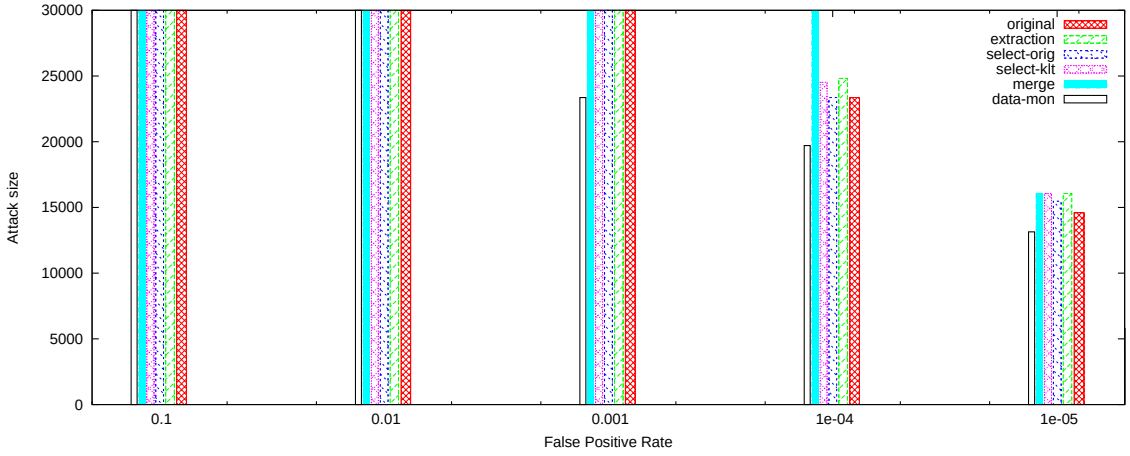


(c) DDK attack

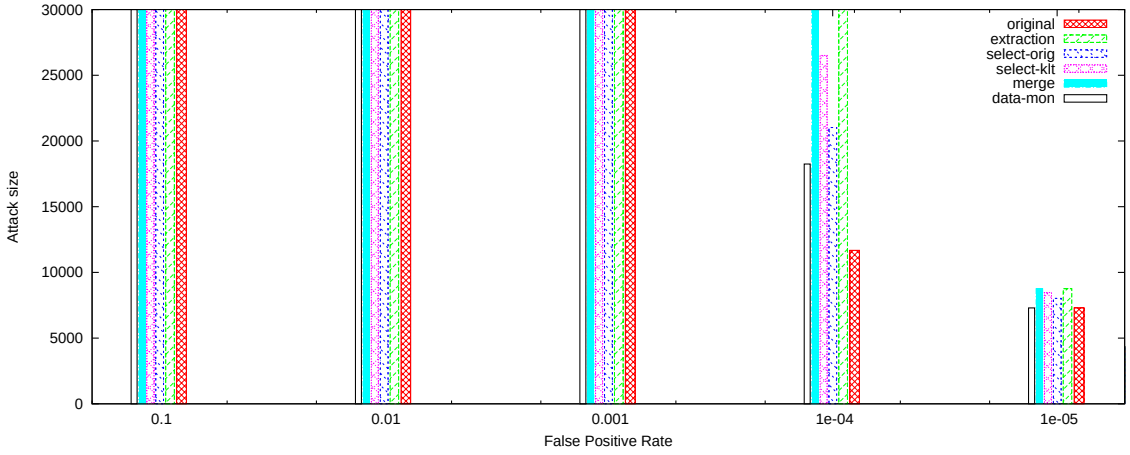
Figure 39: Polymorphic blending attack size for 1-gram PAYL



(a) Windows media services attack



(b) CodeRed attack



(c) DDK attack

Figure 40: Polymorphic blending attack size for 2-gram PAYL

attack instances were able to evade the IDS. As we decrease the false positive rate, IDS increases the error threshold to avoid false positives and consequently it is easier for an adversary to evade the IDS. Therefore, as we decrease the false positive rate, smaller attack sizes are required to avoid detection.

IDS that uses feature extraction, transformed feature selection, and merging are considerably more robust than the original PAYL. The 1-gram features used by PAYL contains many noisy 1-gram and thus it does not perform very well. However, feature extraction, transformed feature selection, and merging, all three remove noisy features and retain only the features that are more helpful in detection. Thus, these three techniques considerably improve the robustness of the original PAYL. Random data monitoring and basic feature selection are very basic techniques and moderately improve the robustness of the IDS. Since both these techniques involve sampling, they might discard potentially important features or data. Therefore, in some occasions, both of these two techniques might reduce the robustness of the IDS.

Blending attack size required to evade improved 2-gram PAYL anomaly IDSs is shown in Figure 40. An optimized and fine tuned original 2-gram PAYL is more effective than 1-gram model in the detection of polymorphic attacks. For 2-gram PAYL, feature extraction, transformed feature selection, and feature merging improves the robustness by using better features.

Since size of attack body and attack vector used by CodeRed is the largest of all three attacks, it requires largest attack size to evade the IDS. On the other hand, windows media services attack, with smallest attack code, requires smallest attack size to escape detection.

The monitoring speed of each IDS was measured. The IDS was deployed on a 2.4Ghz Linux machine with 2GB RAM. Testing data was read from a `tcpdump` file and time taken to classify the complete dataset was recorded. Table 4.6.4 shows the monitoring speed of different IDSs. 1-gram monitoring takes considerably less time

than 2-gram matching. This is due to the following two reasons. 2-gram feature is more complex and takes more time to calculate. Also, the number of unique 2-gram is much higher than number of unique 1-grams (Section 4.2.7).

Basic feature extraction and data sampling methods monitor a sub-sample of space monitored by the original PAYL. Therefore, on an average they are faster than PAYL. In feature merging, IDS merges n-grams using *KLT*. But the extra time for merging is compensated by reduction in the number of features and also the overall speed is faster. Feature extraction and transformed feature selection requires multiplying eigenvector matrix to n-gram features. Hence, they take considerable time to monitor the data.

Table 10: Monitoring speed (in secs/100K packets)

IDS	Original Model	Feature Extraction	Basic Feature Selection	Transformed Feature Selection	Feature Merging	Data Sampling
1-gram	6.33	13.70	4.75	12.67	4.63	6.21
2-gram	786.1	1225	500	1183.6	172	597.6

Considering both robustness and run time efficiency, randomization appears to be the best robustness improvement strategy. Techniques like random feature merging improves the robustness of an IDS and at the same time increases efficiency. Since these randomization techniques sub-sample the monitoring space, there is a risk of increased false negatives. Some of the other countermeasures make IDS very robust, but they also consume more resources and are more complex in design and implementation. In short, trade-offs between robustness and other performance measures need to be carefully considered before choosing a suitable IDS.

4.7 Summary

Existing polymorphic techniques can be used to evade signature-based IDSs because the attack instances do not share a consistent signature. But anomaly IDSs can detect these attack instances because the polymorphism techniques fail to mask their statistical anomalies. We presented a new class of polymorphic attacks called polymorphic blending attacks that overcomes this very shortcoming. The idea is to first learn the normal profiles used by the IDS, and then, while creating a polymorphic instance of an attack, make sure that its statistics match the normal profiles.

We described the basic steps and general techniques that can be used to devise polymorphic blending attacks. We primarily use encryption and padding to modify and generate a blended attack instance that matches the normal profile. We presented a case study using the anomaly IDS `PAYL` to demonstrate that these attacks are practical and feasible. Our experiments showed that polymorphic blending attacks can evade `PAYL` while traditional polymorphic attacks cannot. We also showed that an attacker does not need a large number of packets to learn the normal profile and to blend in successfully.

We presented a formal framework for polymorphic blending attacks. We modeled a variety of IDSs as either FSA or stochastic FSA. We also modeled different polymorphic attack sections using FSA. We determined the complexity of generating different attack sections that match the (s)FSA IDS. Matching attack vector and padding with an FSA IDS can be done in polynomial time using an algorithm similar to *Bellman-Ford* algorithm. The problem of matching attack vector, padding, or attack body with an sFSA IDS was proved to be NP-complete. We presented some techniques to reduce these NP-complete problems to a satisfiability problem or an integer linear programming problem. Thus, optimization algorithms available in these problem domains can be used to generate near-optimal attack sections. We also proposed a heuristic that can be used to find suitable attacks efficiently. We

validated our framework using PAYL 1-gram and 2-gram. The polymorphic blending attacks generated using optimization solvers and heuristics were closely able to match the normal profile. They performed better than the greedy algorithms proposed in the case-study.

We discussed the drawbacks of current anomaly IDSs due to which these polymorphic blending attacks exist. We also proposed techniques to improve the given anomaly IDS so that it becomes harder to generate a blending attack. Using experiments, we showed that the proposed techniques are indeed useful in improving the robustness of anomaly detection systems. However, improving the robustness of an anomaly IDS may also increase the computation complexity of the IDS. Thus the trade-off needs to be considered.

CHAPTER V

CONCLUSION

In this dissertation, we focused on two important aspects of an IDS, namely, run-time efficiency and robustness. The IDS should be efficient to be able to detect intrusions in real-time. Also, the IDS should correctly classify normal activities as normal and attack activities as attack, even in the presence of an adversary who is actively trying to evade it.

To detect intrusions in real-time, the IDS should be fast and should require minimal memory. Since most of the resources in an IDS is typically consumed by the approximate string matching component, an important problem to address is the efficiency of approximate string matching algorithm.

We presented a fast algorithm for an approximate string matching problem, namely q-gram matching, used by various host-anomaly IDSs and network anomaly IDSs. q-gram matching is also used in other applications including, genome sequence matching, misspelling detection, and pattern recognition. To contain false positives, the training dataset of an anomaly IDS should be extensive. For huge text sizes, previous string matching algorithms become unacceptable. Rabin-Karp algorithm works fast for q-gram matching but requires large memory and produces false matches. The large memory requirement of the Rabin-Karp method prohibits the IDS in being deployed inside the kernel where it is suitable to perform system-call based detection.

We presented a tree-based algorithm to perform fast and memory efficient q-gram matching. The text is pre-processed and stored in a tree. Tree pruning was used to reduce the memory requirements and suffix links were added to make the matching fast and efficient. Our experiments showed that the proposed algorithm is

faster and more memory efficient than the previous algorithms. Compared to Rabin-Karp algorithm, the space saving is by an order of magnitude. Space saving is more apparent for high values of q. Since our model is very space efficient, it is suitable for deployment in the kernel. This reduces the time required by the IDS to obtain the monitored data and further improves the overall efficiency of the IDS.

Even though we improved the efficiency of q-gram matching used by various IDSs, other factors may affect the performance of the IDS. Some of these factors are, data acquisition time and latency, alert management, and alert reporting. These factors need to be considered while designing an efficient IDS with real-time detection capabilities. Also, our algorithm is applied to a particular approximate string matching algorithm. As IDSs are evolving, they are using more complex detection models and pattern matching approaches. For example, some IDSs use machine learning techniques like SVM, genetic algorithms, and neural networks for pattern matching. Different run-time optimization techniques and algorithms are required to make these IDSs efficient.

Our algorithm can also be used to improve the efficiency of other applications that use q-gram matching. An interesting future work is to apply our algorithm to other applications and analyse the performance gains.

Ideally, we would like an IDS to detect all the attacks and make no mistakes. However, an IDS typically generates both false positives and false negatives. False negatives or undetected attacks can be particularly high if an adversary is actively trying to evade the IDS. Thus, it is important to determine how robust an IDS is in the face of such active attacks.

We proposed a novel attack called polymorphic blending attack to analyse the robustness of network anomaly IDSs. To generate these attacks, we assume that the adversary knows the IDS training and monitoring algorithm. The adversary first estimates the normal profile (or pattern) used by the IDS. Then the adversary modifies

an initial attack instance so that the final attack instance matches the normal profile. The initial attack instance is modified using different polymorphism techniques including encryption and padding. Using an example IDS, we show that polymorphic blending attacks are indeed feasible.

We presented a formal framework to analyse the complexity of polymorphic blending attacks. We showed that in general, it is NP-complete to generate a PBA that optimally matches the normal profile. However, approximation algorithms proposed in Section 4.4 can be used to generate an attack instance that closely matches the normal profile and can easily evade IDSs.

Using polymorphic blending attacks, we successfully demonstrated that current network anomaly IDSs are not robust against evading attacks. Based on the shortcomings of current anomaly IDSs, we proposed some techniques to improve the robustness of an IDS. Our experiments showed that these techniques are effective in improving the robustness of IDSs.

Our framework considers IDSs that can be represented using (s)FSA. The framework does not incorporate IDSs that use more expressive automatons. Also, modeling of some attack sections, especially decryptor and attack body, is limited. We want to extend our framework to include other IDSs. We would also like to include simple shellcode obfuscation techniques in our framework. Furthermore, we want to include encryption techniques, other than one-to-one byte substitution and XOR, in our framework.

Our proposed framework considers each attack sections separately. A separate algorithm is presented to match each section with the (s)FSA IDS. Another possible approach is to combine models for all the attack sections to create a single model representing the attack packet. Then, the adversary can match the combined model with the normal profile. Compared to the separate model approach, using a combined attack model may generate better matching because, by using the combined attack

model, the adversary can exploit the collective structure of different sections. But matching the combined attack model is considerably more complex than matching the sections separately. An interesting future work is to find an efficient algorithm to match the combined attack model and the normal profile.

5.1 Discussion

The two issues, efficiency and robustness, discussed here are notably dependent on each other. To a certain extent, these two parameters are somewhat conflicting in nature. In Section 4.6, we observed that improving the robustness of an IDS requires better features and consequently reduced efficiency, and vice versa, using simple statistical features for efficiency sacrifices robustness. It is not clear as to what the best trade-off is.

It is hard to find the best compromise because the robustness of an IDS is not easily quantifiable. Some of the criteria of robustness is the size of the attack, the probability of successful evasion, and the amount of time and computing resources used by the adversary. Many of these factors are not measurable. This makes it harder to evaluate the effectiveness of an IDS improvement technique. Other factors, like false positives, false negatives, and IDS management, complicate things further. A unified framework to evaluate an IDS is highly desirable. The framework should take all the important factors related to the IDS operation into consideration. The framework should let users assess the shortcomings of the IDS and suggest steps for rectification.

APPENDIX A

EFFICIENCY

Algorithm 1 q-gram matching using $\mathcal{T}$

for all q-grams, $Q[i, \cdots, i+q-1]$, $i = 1$ to $|Q| - q + 1$ **do**
  Set current node as root of the tree $\mathcal{T}$
  for $j = 0$ to $q - 1$ **do**
    if the current node does not have a child for character $Q[i + j]$ **then**
      reject the current substring $Q[i, \cdots, i+q-1]$ and match the next substring starting from the root node
    else
      traverse to the child node for character $Q[i + j]$
    end if
  end for
  if $j = q - 1$ and we reach a leaf node **then**
    accept substring $Q[i, \cdots, i+q-1]$ and match the next substring starting from the root node
  end if
end for

Algorithm 2 Tree Redundancy Pruning

for all the nodes $Q[i, \cdots, j]$ present in the tree $\mathcal{T}$ **do**
  Compare subtrees rooted at node $Q[i, \cdots, j]$ and its immediate suffix node $Q[i+1, \cdots, j]$
  if the two subtrees are *similar* **then**
    prune the subtree rooted at $Q[i, \cdots, j]$ and make $Q[i, \cdots, j]$ a leaf node
  end if
end for

Algorithm 3 q-gram matching using $\mathcal{T}_p$

for all q-grams, $Q[i, \cdots, i+q-1]$, $i = 1$ to $|Q| - q + 1$ **do**
  Set current node as root of the tree $\mathcal{T}$
  for $j = 0$ to $q - 1$ **do**
    if the current node is a leaf node **then**
      mark the current substring $Q[i, \cdots, i+q-1]$ as *pending* and match the next substring starting from the root node
    else if the current node does not have a child for character $Q[i+j]$ **then**
      for $k = 0$ to $q - j - 1$ **do**
        if $Q[i-k, \cdots, i-k+q-1]$ is marked *pending* **then**
          mark $Q[i-k, \cdots, i-k+q-1]$ as *rejected*
        else
          continue matching the next substring starting from the root node
        end if
      end for
      break
    else
      traverse to the child node for character $Q[i+j]$
    end if
  end for
  if $j = q - 1$ and we reach a leaf node **then**
    mark the current substring $Q[i, \cdots, i+q-1]$ as *accepted* and match the next substring starting from the root node
  end if
end for
mark all the *pending* substrings as *accepted*

Algorithm 4 Adding suffix links in $\mathcal{T}$

for all the nodes $Q[i, \cdots, j]$ present in the tree $\mathcal{T}$ **do**
  create a suffix link from node $Q[i, \cdots, j]$ to its immediate suffix node $Q[i+1, \cdots, j]$
end for

Algorithm 5 q-gram matching using $\mathcal{T}_s$

Set the current node as the root of tree $\mathcal{T}_s$
for $i = 1$ to $|Q| - q + 1$ **do**
  while the current node does not have a child for character $Q[i]$ and we have not reached the root node **do**
    suppose the current node is at depth j
    reject the current substring $Q[i - j, \cdots, i - j + q - 1]$
    traverse the suffix link and match character $Q[i]$ for next substring
  end while
  if the current node is the root node and we still do not have a child for character $Q[i]$ **then**
    character $Q[i]$ is not in the tree, start matching with next character $Q[i + 1]$
  end if
  traverse to the child node for character $Q[i]$
  if the current node is a leaf node **then**
    accept the current substring $(Q[i - q + 1, \cdots, i])$
    traverse the suffix link
  end if
end for

Algorithm 6 Adding suffix links in $\mathcal{T}_p$

for all the nodes $Q[i, \cdots, j]$ present in the tree $\mathcal{T}$ **do**
  for $k = 1$ to $j - i$ **do**
    if node $Q[i + k, \cdots, j]$ exists in $\mathcal{T}_p$ **then**
      create a suffix link from node $Q[i, \cdots, j]$ to it longest suffix node $Q[i + k, \cdots, j]$
    end if
  end for
end for

Algorithm 7 q-gram matching using $\mathcal{T}_{sp}$

Set current node as root of the tree $\mathcal{T}_{sp}$
for $i = 1$ to $|Q| - q + 1$ **do**
  while the current node does not have a child for character $Q[i]$ or we have not reached the root node **do**
    /* A mismatch is found at current node for character $Q[i]$ */
    suppose the current node is at depth j
    for $k = 0$ to $q - j - 1$ **do**
      if substring $Q[i - j - k, \cdots, i - j - k + q - 1]$ is marked *pending* or is unmarked **then**
        mark substring $Q[i - j - k, \cdots, i - j - k + q - 1]$ as *rejected*
      else
        break
      end if
    end for
    traverse the suffix link
  end while
  if the current node is the root node and we still do not have a child for character $Q[i]$ **then**
    character $Q[i]$ is not in the tree, start matching with next character $Q[i + 1]$
  end if
  traverse to the child node for character $Q[i]$
  if the current node is a leaf node **then**
    if the current node is at depth q **then**
      mark the current substring $(Q[i - q + 1, \cdots, i])$ as *accepted*
    else
      mark the current substring $(Q[i - q + 1, \cdots, i])$ as *pending*
    end if
    traverse the suffix link
  end if
end for
mark all the *pending* and unmarked substrings as *accepted*

APPENDIX B

ROBUSTNESS

B.1 Proof of Optimal Padding for 1-gram Blending Attack

We prove that the padding calculated using Equation (15) is minimum for matching the 1-gram profile exactly.

Theorem B.1.1 $\lambda_i \geq 0$, $\forall 1 \leq i \leq c_n$

Proof We prove the theorem by contradiction. Assume that for some j, $\lambda_j < 0$. Then from Equation (15), $\|\hat{w}\|(\delta f(x_j) - \hat{f}(x_j)) < 0$. Thus, $\delta < \frac{\hat{f}(x_j)}{f(x_j)}$. This contradicts Equation (13), therefore, all $\lambda_i \geq 0$. ∎

The frequency of a character x_i in the packet after padding is $\acute{f}(x_i) = \frac{\|\hat{w}\|\hat{f}(x_i)+\lambda_i}{\|\acute{w}\|}$. Using Equation (15) and Equation (12), $\acute{f}(x_i) = f(x_i)$. Thus, the final attack packet after padding has the exact target distribution, $f(x_i)$.

Theorem B.1.2 *The padding calculated using Equation (15) is the minimum required padding to match frequencies exactly.*

Proof Suppose we perform padding using Equation (15). Suppose there exists another packet (say p, $\|p\| < \|\hat{w}\|$) with smaller padding and matches the frequencies exactly. Since $\lambda_k = 0$, the number of occurrences of x_k in p cannot decrease. Thus, frequency of x_k in packet p is $f_p(x_k) = \frac{\|\hat{w}\|\hat{f}(x_k)}{\|p\|} = \frac{\|\acute{w}\|f(x_k)}{\|p\|} > f(x_k)$. Thus, packet p does not match the normal frequencies exactly. Thus, we have reached a contradiction. ∎

B.2 Proof of Hardness of 2-gram Single-Byte Encoding

First, we look at the problem of evading a simple IDS that stores all the 2-grams present in the normal stream. While monitoring, it checks if all the 2-grams present in the traffic are also present in the normal 2-gram list. In the event that the IDS finds a 2-gram that was not present in normal traffic, IDS raises an alarm. Blending the attack packet with the normal traffic requires the attacker to transform the packet such that all the 2-grams in the packet after substitution are also present in the normal 2-gram list. Matching the frequencies of the tuples is at least as hard as the above simplified problem.

Suppose we have a normal traffic profile (N, T_N) and an attack packet description (M, T_M), where N and M is the set of normal and attack characters, respectively. T_N and T_M is the set of different 2-grams present in normal traffic and the attack, respectively. Also, the attacker is allowed to do only one-to-one substitution from M to N. Then, blending of the packet translates to finding a substitution S such that all the tuples in $S(w)$ are also present the normal profile. That is if $a_1a_2 \in T_M$, then $S(a_1a_2) \in T_N$.

Theorem B.2.1 *The problem of finding a one-to-one substitution S to match 2-grams is NP-complete.*

Proof To prove that the problem is in NP-complete, we need to show that the problem is polynomial time verifiable and NP-hard.

Given a solution substitution S for the 2-gram matching problem, we can calculate $S(w)$ in $O(\|w\|)$ steps. For each 2-gram present in $S(w)$, checking if it is present in T_N can be done in $O(\|w\|.T_M)$ steps. Thus, this problem is poly-verifiable and consequently in NP.

To show that the problem is NP-hard, we reduce the problem of sub-graph isomorphism to substitution problem. A sub-graph isomorphism problem is that given

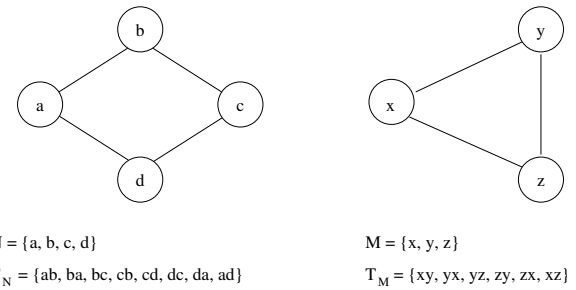


Figure 41: Reducing sub-graph isomorphism to 2-gram matching problem

two graphs $G(V, E)$ and $G'(V', E')$, decide whether G' is a sub-graph of G. Mathematically, we want to check if there is a mapping $S(V' \mapsto V)$, s.t. $\forall(v_1, v_2) \in E', (S(v_1), S(v_2)) \in E$.

Suppose, $N = V$. For each edge $e = (v_1, v_2) \in E$, add two 2-grams (v_1v_2, v_2v_1) in the normal profile (T_N). Suppose $M = V'$. For each edge $e' = (v_1, v_2) \in E'$, we add two 2-grams (v_1v_2, v_2v_1) in the attack profile (T_M).

If the above 2-gram matching problem has a solution, then we can find a mapping $S(V' \mapsto V)$ such that for all 2-grams $(a_1a_2) \in T_M$, $S(a_1a_2) \in T_N$. Since the 2-grams in T_M correspond to edges in G' and the 2-grams in T_N correspond to edges in G, the above statement suggests that $\forall e' \in G', S(e') \in G$. This means that graph G' is isomorphic to a sub-graph of G with mapping given by S.

Also, if there does not exist a solution to the 2-gram matching problem, then there does not exists a substitution S_t such that G' is a sub-graph of G after substitution. Otherwise, S_t will result in a successful 2-gram mapping.

Thus, the 2-gram matching problem is at least as hard as the sub-graph isomorphism problem. It is known that the sub-graph isomorphism problem is NP-complete. Also, we have already proved that the 2-gram matching problem is in NP. Thus, the 2-gram matching problem is NP-complete. ∎

Even if an IDS allows constant number of mismatches, it can be shown that the problem still remains NP-complete. This is followed by the result that sub-graph

isomorphism with constant number of edge insertion, deletion, and substitution is also NP-complete. This means that an attacker cannot get the substitution that will match the normal profile with a small constant number of mismatched 2-grams. Also, the one-to-one substitution problem can be easily reduced to one-to-many substitution. Thus, solving one-to-many substitution is also hard.

B.3 Pseudo-codes for PAYL blending

Algorithm 8 Substitution algorithm for 1-gram blending ($c_m \leq c_n$)

$sort_{desc}(f)$
$sort_{desc}(\acute{f})$
for $i = 1$ to c_m **do**
  $S[y_i] = \{x_i\}$
  $\hat{tf}[i] = f[i]$
end for
for $i = c_m + 1$ to c_n **do**
  minRatio = infinity
  for $j = 1$ to c_m **do**
    ratio = $\frac{\hat{tf}[j]}{\acute{f}[j]}$
    if minRatio > ratio **then**
      label = j
    end if
  end for
  $S[y_{label}] = addToSet(x_i)$
  $\hat{tf}$[label] += $f[i]$
end for
for $i = 1$ to c_n **do**
  $k = S^{-1}[y_i]$ = inverse map of x_i
  prob[y_k is substituted by x_i in attack body] = $\frac{f[i]}{tf[k]}$
end for

Algorithm 9 Substitution algorithm for 1-gram blending ($c_m > c_n$)

n_v = number of vertices in the tree excluding root vertex
$cap[i] = f[i]$
$wt[i]$ = weight of vertex i
$wt_T = \sum_{i=1,c_n} wt[i]$
$wtn[i] = \frac{wt[i]}{wt_T}$ = Normalized weight
$sort_{desc}(wtn)$
for $i = 1$ to n_v **do**
  minDiff = infinity
  for $j = 1$ to c_n **do**
    if minDiff $> |cap[j] - wtn[i]|$ **then**
      setLabel = true
      for all sibling = children of parent[i] **do**
        if label[sibling] == j **then**
          setLabel = false;
        end if
      end for
      if setLabel == true **then**
        minDiff = $|cap[j] - wtn[i]|$
        label[i] = j
        $cap[j] = cap[j] - wtn[i]$
      end if
    end if
  end for
end for

B.4 Hill-Climbing heuristic

Algorithm 10 Pseudo-code for Hill Climbing Heuristic

```
curr_key = generate_random_key()
curr_dist = calculate_distance(curr_key)
best_key = curr_key
best_dist = curr_dist
for MAX times do
  k_i = get_random_key_index()
  min_tmp_dist = INF
  for j in key_range do
    tmp_key = curr_key
    set (k_i = j) in tmp_key
    tmp_dist = calculate_distance(tmp_key)
    if tmp_dist < min_tmp_dist then
      min_tmp_dist = tmp_dist
      new_key_val = j
    end if
  end for
  if min_tmp_dist < curr_dist then
    set (k_i = new_key_val) in curr_key
    curr_dist = min_tmp_dist
  end if
  if curr_dist < min_dist then
    best_key = curr_key
    best_dist = curr_dist
  end if
  if local maxima is reached then
    for i = 1 to 5 do
      k = get_random_key_index()
      v = get_random_key_value()
      set (k = v) in curr_key
    end for
  end if
end for
```

B.5 Matching Attack Vector

Algorithm 11 Pseudo-code for Matching Attack Vector and FSA IDS

FSA_{av} = Attack Vector FSA
$AFSA_{ids}$ = Normal Profile FSA with error states and transitions
$FSA_{int}(\Sigma, Q, q_0, \delta, F) = FSA_{av} \bigcap AFSA_{ids}$
for $i = 0$ to n **do**
  for q in Q **do**
    $dist[q][i] = \infty$
    $prev[q][i] = \phi$
  end for
end for
$dist[q_0][0] = 0$
$reachable[0] = q_0$
for $i = 1$ to l_{av} **do**
  for edge $(q_1, q_2, c) \in F$ **do**
    $alt = dist[q_1][i-1] + cost(q_1, q_2)$
    if $alt < dist[q_2][i]$ **then**
      $dist[q_2][i] = alt$
      $prev[q_2][i] = q_1$
    end if
  end for
end for

REFERENCES

[1] Aho, A. and Corasick, M., "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, pp. 333–340, June 1975.

[2] Akritidis, P., Markatos, E. P., Polychronakis, M., , and Anagnostakis, K., "Stride: Polymorphic sled detection through instruction sequence analysis," *In 20th IFIP International Information Security Conference*, 2005.

[3] Barreno, M., Nelson, B., Sears, R., Joseph, A. D., and Tygar, J. D., "Can machine learning be secure?," *Proceedings of the ACM Symposium on Information, Computer, and Communication Security (ASIACCS)*, 2006.

[4] Boyer, R. and Moore, J., "A fast string searching algorithm," *Communications of the ACM*, vol. 20, p. 762, October 1977.

[5] Burkhardt, S., Crauser, A., Ferragina, P., Lenhof, H., Rivals, E., and Vingron, M., "q-gram based database searching using a suffix array (quasar)," in *RECOMB '99*, pp. 77–83, 1999.

[6] Chang, W. and Marr, T., "Approximate string matching and local similarity," in *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, pp. 259–273, 1994.

[7] Chen, M. and Seiferas, J., "Elegant and efficient subword tree construction," *Combinatorial Algorithms on Words*, pp. 97–107, 1985.

[8] Chinchani, R. and Berg, E., "A fast static analysis approach to detect exploit code inside network flows," *In Recent Advances in Intrusion Detection*, 2005.

[9] Christodorescu, M., Jha, S., Seshia, S., Song, D., and Bryant, R., "Semantics-aware malware detection," *In Proceeding of the IEEE Security and Privacy Conference*, 2005.

[10] Coit, J., Staniford, S., and McAlerney, J., "Towards faster string matching for intrusion detection or exceeding the speed of snort," *DARPA Information Survivability Conference and Exposition (DISCEX II '02)*, vol. 1, pp. 367–373, 2001.

[11] Cole, R. and Hariharan, R., "Approximate string matching: A simpler faster algorithm," *SIAM Journal on Computing*, vol. 31, no. 6, pp. 1761–1782, 2002.

[12] Cormen, T. H., Leiserson, C. E., and Rivest, R. L., "Introduction to algorithms," *The MIT Press/McGraw-Hill*, 1990.

[13] Detristan, T., Ulenspiegel, T., Malcom, Y., and Underduk, M., "Polymorphic shellcode engine using spectrum analysis," *Phrack Issue 0x3d*, 2003.

[14] Duda, R. O., Hart, P. E., and Stork, D. G., "Pattern classification," *John Wiley and Sons*, 2001.

[15] Eckmann, S. T., Vigna, G., and Kemmerer, R. A., "Statl: An attack language for state-based intrusion detection," *JOURNAL OF COMPUTER SECURITY*, vol. 10, pp. 71–104, 2002.

[16] Feng, H., Giffin, J., Huang, Y., Jha, S., Lee, W., and Miller, B., "Formalizing sensitivity in static analysis for intrusion detection," *In Proceedings the IEEE Symposium on Security and Privacy*, 2004.

[17] Feng, H., Kolesnikov, O., Fogla, P., Lee, W., and Gong, W., "Anomaly detection using call stack information," *In Proceedings of the IEEE Security and Privacy Conference*, 2003.

[18] Firew0rker, "Windows media services remote command execution exploit," *http://www.k-otik.com/exploits/07.01.nsiilog-titbit.cpp.php*, April 2005.

[19] Forrest, S., Hofmeyr, S., Somayaji, A., and Longstaff, T., "A sense of self for Unix processes," in *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pp. 120–128, IEEE Computer Society Press, 1996.

[20] Forrest, S., Hofmeyr, S., Somayaji, A., and Longstaff, T., "A sense of self for unix processes," *In Proceedings of the IEEE Symposium on Security and Privacy*, 1996.

[21] Galil, Z. and Giancarlo, R., "Improved string matching with k-mismatches," *SIGACT News*, vol. 17, no. 4, pp. 52–54, 1986.

[22] Group, T. I., "Phatbot trojan analysis," *http://www.secureworks.com/research/threats/phatbot*, June 2007.

[23] Handley, M. and Paxson, V., "Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics," *In 10th USENIX Security Symposium*, 2001.

[24] Karp, R. and Rabin, M., "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Developement*, pp. 249–260, 1987.

[25] Kaufman, C., Perlman, R., and Speciner, M., "Network security: Private communication in a public world," *Prentice Hall*, 2002.

[26] Knuth, D., Morris, J., and Pratt, V., "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 1, pp. 323–360, 1977.

[27] Kolesnikov, O. M., Dagon, D., and Lee, W., "Advanced polymorphic worms: Evading ids by blending in with normal traffic," 2003.

[28] Kruegel, C., Kirda, E., Mutz, D., Robertson, W., and Vigna, G., "Automating mimicry attacks using static binary analysis," *In 14th Usenix Security Symposium*, 2005.

[29] Kruegel, C., Kirda, E., Mutz, D., Robertson, W., and Vigna, G., "Polymorphic worm detection using structural information of executables," *In Recent Advances in Intrusion Detection*, 2005.

[30] Kruegel, C., Toth, T., and Kirda, E., "Service specific anomaly detection for network intrusion detection," *In Proceedings of ACM SIGSAC*, 2002.

[31] Kruegel, C. and Vigna, G., "Anomaly detection of web-based attacks," *In Proceedings of ACM CCS*, pp. 251–261, 2003.

[32] Ktwo, "Admmutate: Shellcode mutation engine," *http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz*, June 2007.

[33] Landau, G. and Vishkin, U., "Efficient string matching with k-mismatches," *Theoretical Computer Science*, pp. 239–249, 1986.

[34] Liang, Z. and Sekar, R., "Fast and automated generation of attack signatures: a basis for building self-protecting servers," *Proceedings of the 12th ACM Conference on Computer and Communications Security (ACM CCS)*, pp. 213 – 222, 2005.

[35] Mahoney, M., "Network traffic anomaly detection based on packet bytes," *In Proceedings of ACM SIGSAC*, 2003.

[36] Mahoney, M. and Chan, P., "Learning nonstationary models of normal network traffic for detecting novel attacks," *In Proceedings of SIGKDD*, 2002.

[37] McCreight, E., "A space-economical suffix tree construction algorithm," *Journal of the ACM*, vol. 23, no. 2, pp. 262–272, 1976.

[38] Navarro, G., "A guided tour to approximate string matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001.

[39] Navarro, G. and Baeza-Yates, R., "Fast and practical approximate string matching," *Information Processing Letters*, vol. 59, pp. 21–27, 1996.

[40] Navarro, G. and Baeza-Yates, R., "Faster approximate string matching," *Algorihtmica*, vol. 23, no. 2, pp. 127–158, 1999.

[41] Newsome, J., Karp, B., and Song, D., "Polygraph: Automatically generating signatures for polymorphic worms," *In Proceeding of the IEEE Security and Privacy Conference*, 2005.

[42] Perdisci, R., Dagon, D., Lee, W., Fogla, P., and Sharif, M., "Misleading worm signature generators using deliberate noise injection," *In Proceedings of the IEEE Security and Privacy Conference*, 2006.

[43] Perdisci, R., Gu, G., and Lee, W., "Using an ensemble of one-class svm classifiers to harden payload-based anomaly detection systems," *ICDM*, 06.

[44] Ptacek, T. and Newsham, T., "Insertion, evasion, and denial of service: Eluding network intrusion detection," *Technical Report T2R-0Y6, Secure Networks, Inc.*, 1998.

[45] Puppy, R. F., "A look at whisker's anti- ids tactics just how bad can we ruin a good thing?," *www.wiretrip.net/rfp/txt/whiskerids.html*, 1999.

[46] Roesch, M., "Snort-lightweight intrusion detection for networks," *In Proceedings of the 13th USENIX conference on System administration*, pp. 229 – 238, 1999.

[47] Rubin, S., Jha, S., and Miller, B. P., "Language-based generation and evaluation of nids signatures," *In Proceeding of the IEEE Symposium on Security and Privacy*, 2005.

[48] Rubin, S., Jha, S., and Miller, B., "Automatic generation and analysis of nids attacks," *In Annual Computer Security Applications Conference (ACSAC)*, 2004.

[49] Schaffer, C., "A conservation law for generalization performance," *International Conference on Machine Learning*, 1994.

[50] Sedalo, M., "Jempiscodes: Polymorphic shellcode generator," *www.shellcode.com.ar/en/proyectos.html*.

[51] Sekar, R., Gupta, A., Frullo, J., Shanbhag, T., Tiwari, A., Yang, H., and Zhou, S., "Specification-based anomaly detection: A new approach for detecting network intrusions," *Proceedings of the 9th ACM conference on Computer and communications security (ACM CCS)*, 2002.

[52] Sekar, R., Gupta, A., Frullo, J., Shanbhag, T., Tiwari, A., Yang, H., and Zhou, S., "Specification-based anomaly detection: a new approach for detecting network intrusions," in *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pp. 265–274, 2002.

[53] Sinz, C., "Towards an optimal cnf encoding of boolean cardinality constraints," *In Principles and Practice of Constraint Programming*, pp. 827–831, 2005.

[54] Song, D., "Fragroute: a tcp/ip fragmenter," *www.monkey.org/∼dugsong/fragroute*, 2002.

[55] Sunday, D., "A very fast substring search algorithm," *Communications of the ACM*, vol. 33, no. 8, pp. 132–142, 1990.

[56] Sutinen, E. and Tarhio, J., "On using q-gram locations in approximate string matching," in *Proceedings of the Third Annual European Symposium on Algorithms*, pp. 327–340, 1995.

[57] Szor, P., "Advanced code evolution techniques and computer virus generator kits," *The Art of Computer Virus Research and Defense*, 2005.

[58] Tan, K., Killourhy, K., and Maxion, R., "Undermining an anomaly-based intrusion detection system using common exploits," *In Recent Advances in Intrusion Detection*, 2002.

[59] Tarhio, J. and Ukkonen, E., "Boyer moore approach for approximate string matching," *Proceedings of SWAT'90*, pp. 348–359, 1990.

[60] Tax, D. M. J. and Muller, K.-R., "Feature extraction for one-class classification," in *ICANN*, pp. 342–349, 2003.

[61] Toth, T. and Kruegel, C., "Accurate buffer overflow detection via abstract payload execution," *In Recent Advances in Intrusion Detection*, 2002.

[62] Ukkonen, E., "Approximate string-matching with q-grams and maximal matches," *Theoretical Computer Science*, vol. 92, no. 1, pp. 191–211, 1992.

[63] Ukkonen, E., "On-line construction of suffix trees," *Algorithmica*, vol. 14, pp. 249–260, 1995.

[64] UNM, "University of New Mexico system call dataset," *http://cs.unm.edu/∼immsec/systemcalls.htm*, June 2007.

[65] Uppuluri, P. and Sekar, R., "Experiences with specification-based intrusion detection," in *RAID '00: Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection*, pp. 172–189, 2001.

[66] Vigna, G., Robertson, W., and Balzarotti, D., "Testing network-based intrusion detection signatures using mutant exploits," *In Proceedings of the ACM Conference on Computer and Communication Security (ACM CCS)*, pp. 21–30, 2004.

[67] Wagner, D. and Dean, D., "Intrusion detection via static analysis," *In Proceeding of IEEE Symposium on Security and Privacy*, 2001.

[68] Wagner, D. and Soto, P., "Mimicry attacks on host-based intrusion detection systems," *In Proceedings of the ACM Conference on Computer and Communication Security (ACM CCS)*, 2002.

[69] Wang, H. J., Guo, C., Simon, D. R., and Zugenmaier, A., "Shield: Vulnerability-driven network filters for preventing known vulnerability exploits," *In the Proceedings of ACM SIGCOMM*, 2004.

[70] Wang, K., Parekh, J. J., and Stolfo, S. J., "Anagram: A content anomaly detector resistant to mimicry attack," *Recent Advances in Intrusion Detection (RAID)*, 2006.

[71] Wang, K. and Stolfo, S., "Anomalous payload-based network intrusion detection," *In Recent Advances in Intrusion Detection*, 2004.

[72] Wang, K. and Stolfo, S., "Anomalous payload-based worm detection and signature generation," *In Recent Advances in Intrusion Detection*, 2005.

[73] Weiner, P., "Linear pattern matching algorithms," *IEEE Symposium on Switching and Automata Theory*, pp. 1–11, 1973.

[74] Wu, S. and Manber, U., "Fast text searching allowing errors," *Communications of the ACM*, vol. 35, pp. 83–91, 1992.

[75] Wu, S., Manber, U., and Myers, E., "A subquadratic algorithm for approximate limited expression matching," *Algorithmica*, vol. 15, no. 1, pp. 50–67, 1996.

[76] Yetiser, T., "Polymorphic viruses: Implementation, detection, and protection," *Technical Report, VDS Advanced Research Group*, 1993.

VITA

Prahlad Fogla is a doctrol candidate in Computer Science at Georgia Institute of Technology. He received his BSc degree (2000) in Computer Science from Indian Institute of Technology and an MSc degree (2003) in Computer Science from Georgia Institute of Technology. His research interests include computer security and worm detection.